1. An inversion in an array $A[1 .. n]$ is a pair of indices $i, j$ such that $i < j$ and $A[i] > A[j]$. The number of inversions in an $n$-element array is between 0 (if the array is sorted) and $\binom{n}{2}$ (if the array is sorted backward). Describe and analyze an algorithm to count the number of inversions in an $n$-element array in $O(n \log n)$ time. **Hint:** Modify mergesort.

**Solution:** Following the hint, we will use a divide-and-conquer approach based on a modified mergesort. We will divide the input array into equal sized subarrays and then conquer both subarrays by recursively sorting the subarrays and computing the number of inversions from pairs of indices sharing a subarray. Finally, we will modify the merge procedure so we both sort the input array and count the number of inversions between pairs of indices lying in different subarrays. The algorithm returns the sum of the recursive counts and the final count done during the merge step.

The procedure $\text{COUNTANDSORT}(A[1 .. n])$ returns the number of inversions in the input array $A$ (before any modifications are done) while also sorting $A$. It relies on a second procedure $\text{COUNTANDMERGE}(A[1 .. n], m)$ that returns the number of inversions between pairs of indices $i \in \{1 \ldots m\}$ and $j \in \{m + 1 \ldots n\}$ while also sorting $A$, assuming $A[1 .. m]$ and $A[m + 1 .. n]$ are already sorted.

```
\text{COUNTANDSORT}(A[1 .. n]):
  if $n \leq 1$
    return 0
  else
    $m \leftarrow \lfloor n/2 \rfloor$
    $count \leftarrow \text{COUNTANDSORT}(A[1 .. m])$
    $count \leftarrow count + \text{COUNTANDSORT}(A[m + 1 .. n])$
    $count \leftarrow count + \text{COUNTANDMERGE}(A[1 .. n], m)$
    return $count$

\text{COUNTANDMERGE}(A[1 .. n], m):
  $count \leftarrow 0$
  $i \leftarrow 1; j \leftarrow m + 1$
  for $k \leftarrow 1$ to $n$
    if $j > n$
      $B[k] \leftarrow A[i]; i \leftarrow i + 1$
    else if $i > m$
      $B[k] \leftarrow A[j]; j \leftarrow j + 1$
    else if $A[i] \leq A[j]$
      $B[k] \leftarrow A[i]; i \leftarrow i + 1$
    else
      $count \leftarrow count + m - i + 1$
      $B[k] \leftarrow A[j]; j \leftarrow j + 1$
  for $k \leftarrow 1$ to $n$
    $A[k] \leftarrow B[k]$
  return $count$
```
We will prove correctness in two stages. First, assume the procedure `COUNTAndMerge` is correct. Procedure `COUNTAndSort` successfully sorts the input array $A[1..n]$, because the sorting code is unchanged from the mergesort we saw in class. If $n \leq 1$, then there are no inversions and the procedure is correct to return 0. Otherwise, assume the procedure is correct when given an array of length $n' < n$. Let $m = \lfloor n/2 \rfloor$. Each inversion $i, j$ in $A$ can be partitioned into three groups; those with $i, j \in \{1, \ldots, m\}$, those with $i, j \in \{m + 1, \ldots, n\}$, and those with $i \in \{1, \ldots, m\}$ and $j \in \{m + 1, \ldots, n\}$. The procedure successfully counts the first two types by our inductive assumption. The other type is counted successfully by our assumption that `COUNTAndMerge` is correct. Procedure `COUNTAndSort` correctly returns the sum of these three counts.

Now, consider running `COUNTAndMerge(A[1 .. n], m)`, and assume $A[1 .. m]$ and $A[m + 1 .. n]$ are both sorted. Again, the sorting steps are unchanged, so it does sort the input array $A[1 .. n]$ correctly. To prove it counts correctly, we will use induction on $n - k + 1$, the number of elements added to $B[k .. n]$ from arrays $A[i .. m]$ and $A[j .. n]$. We will prove that from iteration $k$ onward, the for loop correctly increments `count` by the number of inversions $i', j'$ such that $i \leq i' \leq m$ and $j \leq j' \leq n$. When $k = n + 1$ (i.e., $n - k + 1 = 0$), there are no more inversions to count and the loop correctly terminates. Otherwise, assume the algorithm increments `count` correctly for iterations $k'$ onward for $k' > k$ (i.e., $n - k' + 1 < n - k + 1$). The first two cases of the if-else chain are cases where there are no more pairs $i', j'$ with both $i'$ and $j'$ in the above ranges, so the procedure is correct not to increment `count`. In the third case, $A[i] \leq A[j']$ whenever $j \leq j' \leq n$, because $A[m + 1 .. n]$ is sorted. Therefore, $i$ is not in any more inversions, and it can be safely ignored in future iterations. Finally, in the last case $A[j] < A[i']$ whenever $i \leq i' \leq m$. The procedure is correct to increment `count` by the number of such $i'$, $m - i + 1$. Having counted all the inversions with $j$, the algorithm is correct to ignore it in future iterations. The future iterations continue to increment `count` correctly by our inductive assumption.

`COUNTAndMerge` has the one $\Theta(n)$ time for loop, and the rest of `COUNTAndSort` outside the two recursive calls takes constant time. Letting $T(n)$ be the running time of `COUNTAndSort(A[1 .. n])` and ignoring floors, we have $T(n) = 2T(n/2) + \Theta(n)$ which we know solves to $T(n) = \Theta(n \log n)$. ■

**Rubric:** 10 points total: 5 points for the algorithm, 3 point for the proof, 2 points for the running time analysis.
2. Funtime FunDollars come in only three denominations: 1FD bills, 4FD bills, and 6FD bills. As an employee of Funtime, you will need to make change (in FunDollars) for guests.

(a) Consider the following greedy algorithm for making change for \( k \) FunDollars. Hand over the largest bill less than or equal to \( k \), and then recursively make change for the amount that remains. Give an example where this strategy forces you to hand over more bills than the minimum possible.

**Solution:** Suppose we try making change for 8 FunDollars. The greedy strategy will use one 6FD bill and two 1FD bills for three bills total. However, we could hand over two 4FD bills instead.

**Rubric:** 2 points total: -1 for not describing a better solution than the greedy one.

(b) Design a recursive algorithm that computes, given a value \( k \), the minimum number of bills needed to make change for \( k \) FunDollars. You do not need to worry about making your algorithm efficient, but it should be correct. You should express the running time of your algorithm as a recurrence in \( k \), but you do not have to solve the recurrence.

**Solution:** The following algorithm will work based on the following observation: after choosing our first bill to hand over, the optimal way to make change is to use the minimum number of bills for what remains. The procedure \( FDR\text{Rec}(k) \) returns the minimum number of bills needed to make change for \( k \) FunDollars.

\[
FDR\text{Rec}(k):
\begin{align*}
\text{if } k &= 0 \\
\quad \text{return } 0 \\
\text{else} \\
\quad &\quad \text{bills} \leftarrow 1 + FDR\text{Rec}(k - 1) \\
\quad &\quad \text{if } k \geq 4 \\
\quad &\quad \quad \text{bills} \leftarrow \min\{\text{bills}, 1 + FDR\text{Rec}(k - 4)\} \\
\quad &\quad \text{if } k \geq 6 \\
\quad &\quad \quad \text{bills} \leftarrow \min\{\text{bills}, 1 + FDR\text{Rec}(k - 6)\} \\
\quad \text{return bills}
\end{align*}
\]

The procedure is correct for \( k = 0 \), because there are no bills to hand over in that case. For larger \( k \), we may assume the procedure correctly minimizes the number of bills used when making change for \( k' < k \) FunDollars. Per the observation, the procedure correctly returns the minimum over all feasible choices of first bill and optimal ways to make change with what remains.

Let \( T(k) \) be the running time of \( FDR\text{Rec}(k) \). The procedure makes up to three recursive calls and does a constant amount of work otherwise, so in the worst case

\[
T(k) = T(k - 1) + T(k - 4) + T(k - 6) + \Theta(1).
\]

(This recurrence solves to \( T(k) = \Theta(q^k) \) where \( q \approx 1.4656 \). Fortunately, we’ll do much better in the next part.)

**Rubric:** 3 points total: 2 points for the algorithm and proof, 1 point for the recurrence.

(c) Design and analyze an efficient algorithm that computes, given a value \( k \), the minimum number of bills needed to make change for \( k \) FunDollars. Your analysis should give the asymptotic running time of your algorithm in terms of \( k \). This algorithm should be fast.
Solution: For simplicity, let’s rewrite the algorithm from part (b) as a recurrence. For any integer \( i \) such that \( 0 \leq i \leq k \), let \( FD(i) \) be the minimum number of bills needed to make change for \( i \) FunDollars. We want to compute \( FD(k) \). The observation from part (b) immediately yields the following recurrence:

\[
FD(i) = \begin{cases} 
0 & \text{if } i = 0 \\
1 + FD(i-1) & \text{if } 1 \leq i < 4 \\
1 + \min\{FD(i-1), FD(i-4)\} & \text{if } 4 \leq i < 6 \\
1 + \min\{FD(i-1), FD(i-4), FD(i-6)\} & \text{otherwise}
\end{cases}
\]

There are \( k + 1 \) recurrence subproblems, and each one takes \( O(1) \) time to compute given the dependent subproblem answers, so we should be able to compute \( FD(k) \) using \( O(n) \) space and time. We will store solutions to recurrence subproblems in an array \( FD[0..k] \). For the base case, we fill in \( FD[0] \leftarrow 0 \). Each entry is dependent upon up to three entries to the left (entries of lower index), so we fill in the array from left to right (from low index to high).

The above is enough for a perfect score. But for completion’s sake, let’s write out an iterative procedure \( FD\text{Iter}(k) \) for computing the minimum number of bills needed to make change for \( k \) FunDollars.

\[
\begin{align*}
FD\text{Iter}(k): \\
&FD[0] \leftarrow 0 \\
&\text{for } i \leftarrow 1 \text{ to } k \\
&\quad bills \leftarrow 1 + FD[k-1] \\
&\quad \text{if } k \geq 4 \\
&\quad \quad bills \leftarrow \min\{bills, 1 + FD[k-4]\} \\
&\quad \text{if } k \geq 6 \\
&\quad \quad bills \leftarrow \min\{bills, 1 + FD[k-6]\} \\
&\text{return } FD[k]
\end{align*}
\]

Rubric: 5 points total: 3 points for filling an array, 2 points for running time analysis. Part (b) or a recurrence given in this part must be correct to receive any credit.