Describe and analyze an algorithm to compute the weighted median of a given weighted set in $O(n)$ time. Your input consists of two unsorted arrays $S[1..n]$ and $W[1..n]$, where for each index $i$, the $i$th element has value $S[i]$ and weight $W[i]$. You may assume that all values are distinct and all weights are positive.

**Solution:** Taking a cue from the standard median problem, we will describe a procedure $\text{WeightedSelect}(S[1..n], W[1..n], goal)$ that takes a set of values, a set of weights, and a goal weight $goal \leq w(S)$. Like $\text{MomSelect}$, the procedure may move some elements around in $S$ and $W$, but it then returns the index of the least value element $x$ such that $w(S_{\leq x}) < goal$ and $w(S_{< x}) + w(x) \leq goal$. This element must exist, because of our guarantee that $goal \leq w(S)$. Because values are distinct and weights are positive, it must also be the case that $w(S_{\geq x}) \geq w(S) - g$ for this particular element $x$. Our algorithm naturally returns the index from calling $\text{WeightedSelect}(S[1..n], W[1..n], w(S)/2)$.

Part of the procedure is a call to the subroutines $\text{MomSelectPlus}(S[1..n], W[1..n], k)$ and $\text{PartitionPlus}(S[1..n], W[1..n], p)$. These $O(n)$ time routines modify $S$ in the same manner and have the same return values as the standard $\text{MomSelect}$ and $\text{Partition}$ we saw in class, but they also move the same elements around in $W$ to keep the indices between $S$ and $W$ consistent. Here’s the final psuedocode for $\text{WeightedSelect}$.

```python
def WeightedSelect(S[1..n], W[1..n], goal):
    if n == 1
        return 1
    else:
        p ← MomSelectPlus(S[1..n], W[1..n], ⌊n/2⌋)
        r ← PartitionPlus(S[1..n], W[1..n], p)
        smaller ← 0
        for i ← 1 to r - 1
            smaller ← smaller + W[i]
        if goal < smaller
            return WeightedSelect(S[1..r - 1], W[1..r - 1], goal)
        else if goal > smaller + W[r]
            return WeightedSelect(S[r + 1..n], W[r + 1..n], goal - smaller - W[r])
        else
            return r
```

If $n = 1$, then we’re correct to return the only element’s index. Otherwise, we’re selecting a pivot element $x$ (using $\text{MomSelectPlus}$), and partitioning the arrays so everything in $S_{\leq x}$ has smaller index and everything in $S_{> x}$ has larger index. Variable $smaller$ is set to $w(S_{\leq x})$ by the for loop. Now, if $goal < smaller$, then $w(S_{\leq x})$ is too big for $x$ to be the right element to return. We recursively search for the goal element in $S_{< x}$ (and the recursive call is correct by induction on $n$). Otherwise, if $goal > smaller + W[r]$, then the value of $x$ is too small, and we need to search $S_{> x}$. However, we need to reduce the goal weight in the recursive call to reflect the amount of weight in lessor valued items outside the recursive call. Finally, if neither condition passes, then $r$ must be the index of the element we’re looking for and we’re right to return it.

The time to do everything outside the recursive calls, including the calls to $\text{MomSelectPlus}$, $\text{PartitionPlus}$, and the for loop is $O(n)$. Because $\text{MomSelectPlus}(S[1..n], W[1..n], ⌊n/2⌋)$
returns the median valued element, both potential recursive calls work on arrays of length at most \( \lfloor n/2 \rfloor \). We make exactly one of these recursive calls, so we can express the running time using the recurrence \( T(n) \leq T(n/2) + n \). Using recursion trees, we can easily verify the running time is \( O(n) \).

\[ \Box \]

**Rubric:** 10 points total: 5 points for the algorithm, 3 points for the justification, 2 points for the running time analysis.
(a) Give an example where this greedy algorithm uses more Dream Dollar bills than the minimum possible.

**Solution:** Suppose we are asked to give change for $416. The greedy algorithm will use seven bills: $365 + $28 + $13 + $7 + $1 + $1 + $1. However, we can get away with five bills: $91 + $91 + $91 + $91 + $52.

The other counterexamples less than $1000 are $455, $507, $546, $598, $637, $689, and $728. This list should be exhaustive (unless Kyle writes buggy code).

**Rubric:** 4 points total: -2 for not describing a better method to make change than what greedy achieves.

(b) Describe a recursive algorithm that computes, given an integer $k$, the minimum number of bills needed to make $k$ Dream Dollars. *Express your running time using a recurrence relation.*

**Solution:** We'll use a backtracking algorithm.
No bills are needed when making change for $0. When $k > 0$, the algorithm tries to decide on a first bill from all the choices that are smaller than the target amount. For each possible choice, it recursively computes the optimal way to make the remaining change, leading to a total of 1 bill plus however many are needed in the recursive call. It finally returns the minimum amount from all the attempts.

Let $T(k)$ be the time needed to run the algorithm on input $k$. When $k < 365$, the algorithm takes constant time, so $T(k) = \Theta(1)$ in these cases (yes, the constant may be very large). Otherwise, it makes eight recursive calls on problems of different inputs and does a constant amount of extra work, leading to the conclusion that

$$T(k) = T(k-1)+T(k-4)+T(k-7)+T(k-13)+T(k-28)+T(k-52)+T(k-91)+T(k-365)+1.$$
(a) Describe a simple recursive algorithm to compute, given two sequences \(A[1..m]\) and \(B[1..n]\), the length of the longest common subsequence of \(A\) and \(B\).

**Solution:** We call the backtracking procedure \(LCS(A[1..m],B[1..n])\) described below.

```
LCS(A[1..m],B[1..n]):
    if m = 0 or n = 0
        return 0
        return 1 + LCS(A[2..m],B[2..n])
    else
        return max(LCS(A[2..m],B[1..n]),LCS(A[1..m],B[2..n]))
```

If \(m = 0\) or \(n = 0\), then every subsequence of either \(A\) or \(B\) is empty, including the longest common one between them, so we return 0. Otherwise, both \(m \geq 1\) and \(n \geq 1\). If \(A[1] = B[1]\), then the longest subsequences common to both should take their first characters from \(A[1]\) and \(B[1]\). If take their first characters of the subsequences from neither the first character of \(A\) or \(B\), we could always extend the two (common) subsequences by adding \(A[1]\) and \(B[1]\) to their beginning. If they use only \(A[1]\) (resp. \(B[1]\)), then we could replace the first character of the other by pulling from \(B[1]\) (resp. \(A[1]\)) instead. The rest of common subsequence should be one that’s long as possible from what remains of \(A\) and \(B\), and the recursive call correctly finds it by induction on \(m + n\).

Finally, if \(A[1] \neq B[1]\), then one of those two characters cannot be used for the beginning of the longest common subsequence. Once discarding one of them, we need the longest common subsequence of what remains from \(A\) and \(B\), which the recursive call finds by induction on \(m + n\). We take the better of the choices of discarding \(A[1]\) and \(B[1]\) respectively using the max.

**Rubric:** 5 points total: 3.5 points for the algorithm, 1.5 points or the justification. -1 for incorrect base case. -1.5 for adding the wrong values to recursive solutions. No running time analysis is needed. Any correct backtracking algorithm that could reasonably be made \(O(n^2)\) time using dynamic programming is worth full credit.

(b) Describe a simple recursive algorithm to compute, given two sequences \(A[1..m]\) and \(B[1..n]\), the length of the shortest common supersequence of \(A\) and \(B\).

**Solution:** We call the backtracking procedure \(SCS(A[1..m],B[1..n])\) described below.

```
SCS(A[1..m],B[1..n]):
    if m = 0
        return n
    else if n = 0
        return m
        return 1 + SCS(A[2..m],B[2..n])
    else
        return min(1 + SCS(A[2..m],B[1..n]),1 + SCS(A[1..m],B[2..n]))
```
If \( m = 0 \), then the shortest common supersequence needs to include all \( n \) characters of \( B \) and nothing else. Similarly, if \( n = 0 \), it needs to include the \( m \) characters of \( A \). Otherwise, there are characters in both \( A \) and \( B \). If \( A[1] = B[1] \), then the first character of the common supersequences of \( A \) and \( B \) should come from both \( A[1] \) and \( B[1] \). If it came from only one of them, we could rearrange things so we use the first character from both. After “covering” \( A[1] \) and \( B[1] \), we still need a minimum number of characters to cover what remains of \( A \) and \( B \), which we find during the recursive call by induction on \( m + n \).

Finally, if \( A[1] \neq B[1] \), then one of those two characters cannot be used as the first character of the shortest common supersequence. After using one of them, we need the shortest common supersequence of what remains from \( A \) and \( B \), which the recursive call finds by induction on \( m + n \). We take the better of the choices of \( A[1] \) and \( B[1] \) respectively using the \( \min \).

**Rubric:** 5 points total: 3.5 points for the algorithm, 1.5 points or the justification. -1 for incorrect base case. -1.5 for adding the wrong values to recursive solutions. No running time analysis is needed. Any correct backtracking algorithm that could reasonably be made \( O(n^2) \) time using dynamic programming is worth full credit.