Chapter 6

19. Maintain a two-dimensional array A that records whether or not a string of length n is an interleaving of x and y, where one dimension correspond to the number of characters in the x repetition, and the other dimension is the number of characters in the y repetition. We also extend x and y using repetition to each be at least n characters.

    Note that if s is a interleaving, then the last character belongs to either x or y. Suppose that the final interleaving uses i characters of x, and j characters of y. Since the total length must be n, our algorithm can only be valid for $i + j = n$. If the last character belongs to x, then there must be some $i' < i$ such that $A[i-1][j]$ is valid, and $x[1 : i'] + s[n] = x[1 : i]$, where the second condition is the verification that the nth character is actually the next character in the repetition of x. If the last character belongs to y, then there must be some $j' < j$ such that $A[i][j-1]$ is valid, and $y[1 : j'] + s[n] = y[1 : j]$. We have $A[i][j]$ is True if and only if one of these two conditions is True. Note that instead of checking that $x[1 : i'] + s[n] = x[1 : i]$, it suffices to show that $s[n] = x[i]$, since we know that if $A[i-1][j]$ is valid, then the first i' characters line up correctly. Similarly, we only have to check $s[n] = y[j]$.

    Finally, we can construct our algorithm, starting from the initial condition that $A[0][0] = True$, since any two strings form an interleaving of 0 characters.

```
def PROBLEM-19:
    A[0][0] = True
    Repeat x until length is at least n
    Repeat y until length is at least n
    for length in 1,...,n:
        for i in 0,...,length:
            j = length - i
            if A[i-1][j] == True and s[length] == x[i]:
                A[i][j] = True
            elif A[i][j-1] == True and s[length] == y[j]:
                A[i][j] = True
            else:
                A[i][j] = False

    for i in 0,...,n:
        if A[i][n - i] == True:
            return True
    return False
```

Chapter 5

2. We can use a similar divide-and-conquer algorithm to find the number of significant inversions.

   Divide: Split the list in two, and recursively find the number of significant inversions of each of the two halves.

   Conquer: The total number of inversions in the original list is the sum of the inversions in the two sub-lists and the number of inversions between the two halves. We can assume that the two sub-lists are sorted, since when we combine them, we end up combining the sub-lists into one sorted list.

   To find the number of inversions between the two sorted halves, we can proceed with a normal merge, comparing the smallest elements of both lists. When comparing the two elements, we compare the left element with twice the right element. If the left is greater, then the remaining elements of the right are also inversions, since the list is sorted. Since we iterate through the n elements to merge, this counting can be done in $O(n)$ time.

   If $T(n)$ is the amount of time necessary to count the number of significant inversions on an n element list, then $T(n)$ follows the following recursive relation:

   $$T(n) = 2T(n/2) + cn$$

   By the master theorem, the runtime of this algorithm is $O(n \log n)$.

3. We design an algorithm that determines if there is a set of more than $n/2$ equivalent cards as well as the account associated with them, if applicable.

   Divide: Split the cards into two piles, and recursively determine if there is a set of more than $n/4$ equivalent cards on the left, and a set of more than $n/4$ equivalent cards on the right.

   Conquer: There are three cases to consider.

   (a) If neither of the two piles has $n/4$ equivalent cards, then there cannot be a set of $n/2$ equivalent cards, and we can simply return No.

   (b) Suppose exactly one of the two piles has more than $n/4$ equivalent cards. Let the account associated with these equivalent cards be A. Because we know the number of cards equivalent to A in one of the piles, we can simply compare A to each of the cards in the other pile. If the sum of the number of equivalent cards in both piles exceeds $n/2$, then we can return Yes, and the account associated with the card. Note that this is the only way we can get a set of more than $n/2$ equivalent cards, since the number of equivalent cards is at most $n/4$ in both piles, and $n/2 < n/2 + 1$. Because there are $n/2$ cards in the other pile, this operation requires $O(n)$ invocations of the equivalence tester.

   (c) Suppose both of the piles have $n/4$ equivalent cards. Similar to the previous case, we know the account associated with these equivalent cards, and we can simply scan the other pile for the number of cards equivalent to that card. If this sum is greater than $n/2$, then we can return Yes and the account associated with the card. If the sum is not greater than $n/2$, then we repeat the process with the other pile, scanning the first pile for cards equivalent to the other account. If either of these two sums are greater than $n/2$, then we can return Yes and the corresponding card. Otherwise, we can return No, as there are only $n/4 + n/4 - 2$ remaining cards, which cannot exceed $n/2$. Because we scan twice, and each scan requires $O(n)$ invocations of the equivalence tester, this operation also requires $O(n)$ invocations of the equivalence tester.

   If $T(n)$ is the number of equivalence tester calls necessary to determine if there are more than $n/2$ equivalent cards and the associated account, then $T(n)$ follows the following recurrence relation:

$$T(n) = 2T(n/2) + cn$$

   By the master theorem, this algorithm requires $O(n \log n)$ calls to the tester.

5. We divide the list of lines into two sublists, recursively find the visible lines in each half, then merge the solutions together to find the total number of visible lines. In addition, our algorithm will compute a list of points, where point p_i is the point of intersection between lines l_i and l_{i+1}.

   Divide: We sort the list in order of increasing slope, then split the list into two equal halves along the median. We recursively find all of the visible lines on the left half, and all of the visible lines on the right half. Let $\mathcal{L}_1$ be the first half, and $\mathcal{L}_2$ be the second half.

   Conquer: Let $\ell = \{\ell_1, ..., \ell_i\}$ be the set of visible lines found by recursively calling the routine on $\mathcal{L}_1$ and $l = \{l_1, .., l_j\}$ be the set of visible lines found by recursively calling the routine on $\mathcal{L}_2$. Note that ℓ and l are sorted in order of increasing slope. Let $A = \{a_1, ..., a_{i-1}\}$ be the set of intersection points, where a_p is the intersection between ℓ_p and ℓ_{p+1}. Similarly, let $B = \{b_1, ..., b_{j-1}\}$ be the set of intersection points among l. Because each ℓ is a visible line and are sorted by increasing slope, the x-coordinates of A are also in sorted order. Similarly, the x-coordinates of B are in sorted order. Therefore, we can merge these two lists into a single sorted list $C = \{c_1, ..., c_{i-j-2}\}$ in linear time. For each point in C, there is a line ℓ_s in ℓ with the highest y-coordinate at that point, and a line l_t in l with the highest y-coordinate at that point, although which of the two lines is higher is not necessarily known. Suppose that there exists points where l_t has a higher y-coordinate than ℓ_s. We consider the point with the smallest such property c_k, and we let (x', y') be the point where ℓ_s and l_t intersect. Because c_k is the smallest point where $l_t > \ell_s$, we know that for x smaller than c_k, $\ell_s > l_t$. Furthermore, because these were defined to be the highest lines at c_k, ℓ_s is higher than all other $l_1, ..., l_{t-1}$ at $x' - \epsilon$, and l_t is higher than all other $\ell_{s+1}, ..., \ell_i$ at $x' + \epsilon$. Therefore, these lines are no longer visible.

   Therefore, we know that the sequence of visible lines in the merged lists is $\ell_1, ..., \ell_s, l_t, ..., l_j$. Furthermore, by definition of (x', y'), this is the intersection point between the new neighboring lines, so we have the sequence of intersection points in the merged lists is $a_1, ..., a_s, (x', y'), b_t, ..., b_j$. Since we have merged the two lists in $O(n)$ time, we can recursively call this function to finish the algorithm.

   If $T(n)$ is the amount of time necessary to find the visible lines out of a set of n lines, then $T(n)$ follows the following recursive relation:

   $$T(n) = 2T(n/2) + cn$$

   By the master theorem, this algorithm runs in $O(n \log n)$ time.