Jason Lai
204995126
Dis 1B

CS 180
May 23, 2019

Homework 6

1 Problem 6.19

Give an efficient algorithm that takes strings s, x, and y and decides if s is an interleaving of x and y.

An efficient algorithm that takes strings s, x, and y and decides if s is an interleaving of x and y is as follows:

```
Let s be the signal of length n that we are interested in
Let p be a repetition of length n of the signal from ship 1
Let q be a repetition of length n of the signal from ship 2
getInterleaving(s, p, q) {
  Array DP[n,n]
  Set DP[0,0]=valid
  For i=1 to n {
    For all pairs (j, k) where j + k = i {
      If DP[j-1,k]=yes and s[k]=p[i]
        DP[j,k]=valid
      Else if DP[j,k-1]=yes and s[k]=q[i]
        DP[j,k]=valid
      Else
        DP[j,k]=invalid
    }
  }
  For all j,k such that j+k=n {
    If DP[j,k]=valid
      Return valid
  }
}
```

From the problem, it's understood that we want to know if the signal s of length n is an interleaving of the repetitions of the signals x and y from two ships, wherein each symbol s_i of s belongs to either x or y. However, this also entails that the repetition of x and y could be of different lengths in the signal s and each symbol s_i could belong to either x or y, which means all combinations could be considered. Here, this algorithm builds the 2-D dynamic programming array each time we consider a symbol s_i to be either from x or y. First, we initialize two signals p and q to be repetitions of length n of x and y respectively. This handles the case that s may solely be either signal, but also that x and y doesn't have to start from the beginning each time they reach their end when checking for repetitions. In the algorithm, as we iterate through s, at symbol s_i, this means there are i symbols up to this point, so there could be any number j of symbols from x and any number k of symbols from

y in s up to that point, such that $j+k=i$. So the algorithm must check if either the previous element using x or y as the repetition is valid, to account for all possible combinations; if one is valid, and the current symbol s_i is a valid next symbol in the repetition for that respective signal, then we set this element in the DP array to be valid and continue. If neither of the previous elements of the 2-D DP array (the one for x or y) was valid, then we say that this possible combination for interleaving is invalid. At the end of the algorithm, if an element in the DP array for j, k such that $j+k=n$ is valid, then we say that the signal s is an interleaving of repetitions of x and y. The time complexity of the algorithm is $O(n^2)$, as there is a single nested loop and a secondary unnested loop, where all loops iterate the length n of s, so the complexity is $O(n^2+n) \approx O(n^2)$.

2 Problem 5.2

Give an O(n log n) algorithm to count the number of significant inversions between two orderings.

A divide and conquer algorithm to count the number of significant inversions between two orderings is as follows:

Let A be the set of n numbers a_1, a_2, ..., a_n where n>1

```
Let k = ceil of n/2
Let S_1 be half of A of size k:  a_1, a_2, ..., a_k
Let S_2 be the other half of A of size n-k:  a_{k+1}, a_{k+2}, ..., a_n
getSortedSet(S) {
  Let Sort_count=0
  Starting from the end of the set S, recursively sort S
    Compare each number i from the end of the set to all earlier elements
      If i is greater than an element j
        If i > 2j
          Increment Sort_count
        Continue to compare i to earlier elements
      If i is less than or equal to an element j
        Place i at the index after that element j
  When S is fully sorted, return Count and S
}
Pass the sets S_1 and S_2 to the above function and recursively call it
Once we have S_1 and S_2 sorted, pass them to the next function
getMergedInversions(S_1, S_2) {
  Let Merge_count=0
  Let M be an empty set
  For all pairs of numbers (a_i, a_j) where a_i ∈ S_1 and a_j ∈ S_2, starting from
  the first elements in S_1 and S_2 (keep track with pointers)
    If a_i > a_j
      If a_i > 2a_j
        Increment Merge_count=0
      Append a_j to M
    Else
      Append a_i to M
  Return Merge_count and M
}
Return Final_count = Sort_count_{S_1} + Sort_count_{S_2} + Merge_count
```

Here, the algorithm behaves just like the inversion count algorithm given in the chapter: a set A of n numbers a_1, a_2, ..., a_n is divided in half, each half is then recursively sorted and the significant inversions in each are counted as they are sorted, then, when the two lists are sorted, they are merged. In the merging process, pairs of numbers, one from each set, are compared and the smaller number or number with the earlier precedence if they are equal is then appended to a final sorted set, and significant inversions are counted during the comparisons. We use pointers, as specified in the book, to keep track of what elements in either set we are looking at. The only change here is that, during sorting or merging, when we compare numbers to be greater than or less than, we have a secondary comparison that compares the number with earlier precedence to be greater than twice the second number; this handles counting the actual number of significant inversions to be summed in the final count. The running time of this algorithm is $O(n\log n)$, as it behaves just like the Merge-and-Count algorithm for regular inversion counts, with only the extra $O(1)$ operation to count significant inversions.

3 Problem 5.3

Their question is the following: among the collection of n cards, is there a set of more than n/2 of them that are all equivalent to one another? Show how to decide the answer to their question with only O(n log n) invocations of the equivalence tester.

A divide and conquer algorithm to decide whether the is a set of more than $n/2$ equivalent cards in a collection of n cards that runs in $O(n\log n)$ time is as follows:

Let C be the collection of n cards.

```
Let S1 be half of C of size ceil of n/2
Let S2 be the other half of C of size floor of n/2
getEquivalentCard(S) {
  If there are no elements in S
    Return nothing
  Else if there is only one element in S
    Return that card in S
  Else if there's at least two cards in S
    If the first two cards are equivalent
      Return either card
    Else
      Return nothing
}
First pass S1 to the above function
If the function returns a card
  Compare the card to all other cards
    If the card is a majority-equivalent card, then return this card
Else
  Remove the first element of S1 and call the algorithm again
If no card in S1 was found to be majority-equivalent
  Repeat the same process for S2
```

In this algorithm, we take advantage of the fact that, if there is an account for which $>n/2$ cards are for that account, then if we divide the collection C in half, then either of the halves S_1 or S_2 will return a card of that equivalence class from the algorithm, as either of the halves has a majority of its cards belonging to that equivalence class. The algorithm will work like this: once the collection C is divided in half, pass one half S_1 to the recursive function. The recursive function will take in the data structure of the set and compare the first two elements of the set. If they are equivalent, then it returns one of the cards to be compared to the rest of the cards in C. Otherwise, if they were not equivalent, the algorithm

will recursively compare elements in the set after removing the first element, until either one (if size of S_1 is odd) or no elements are left in S_1. If a card of majority-equivalence was not found, then the algorithm will run for S_2 and do the same thing. The runtime for this algorithm runs in $O(n\log n)$, as the algorithm satisfies the equation $T(n) \leq 2\,T(n/2) + cn$, wherein there are at most 2 recursions through the algorithm constituting the $2\,T(n/2)$ and at most $2n$ iterations when checking a card returned by the recursions against all other cards, so $c = 2$.

4 Problem 5.5

Give an algorithm that takes n lines as input and in O(n log n) time returns all of the ones that are visible.

A divide and conquer algorithm that takes n lines as input and in $O(n \log n)$ time returns all of the ones that are visible is as follows:

Let L be the set of n nonvertical lines in the plane.

```
Let k = ceil of n/2
Let R1 be half of L of size k:  l1, l2, ..., lk
Let R2 be the other half of L of size n-k:  lk+1, lk+2, ..., ln
Sort the lines in R1 and R2 by increasing slope
  Compute the intersection points of contiguous lines
Let M be the set of intersection points of contiguous lines in R1
Let N be the set of intersection points of contiguous lines in R2
mergeVisibleLines(M, N) {
  Let S be an empty set
  For all pairs of x-coordinates (ai, aj) where ai ∈ M and aj ∈ N, starting
  from the first elements in M and N (keep track with pointers)
    If ai > aj
      Append aj to S
    Else
      Append ai to S
  Return S
}
Traverse through the set of intersection points for the respectively lines
  Compare intersection points between the lines
  If a line covers the others, add the line to a set F and remove it
Return F
```

Here, the algorithm (very rough idea) takes advantage of the fact that, since it's given that no more than two lines may intersect at a point, then between two lines l_i and l_j where their slopes $s_i < s_j$, the region to the left of their intersection point is where l_i is visible and the region to the right of the intersection point is where l_j is visible. From this intuition, we know that for at most 2 lines, these lines are visible if they're the only ones in the set, and for three lines, a line is the uppermost if it intersects the first line (with lowest slope) left of the intersection point of the third line (with highest slope) with the first line; this is how we can tell if a line covers the other lines. So the algorithm will divide the set L of n lines in half, then for each half, recursively sort the lines by increasing slope, all the while computing

the intersection points of contiguous lines in the subsets. At the end, merging the two sets by the order of x-coordinates of the intersection points then, traversing the final set of lines, we check to see if a line at an intersection point hides the other lines by (known by its slope). If it does hide the other lines, it is uppermost so we add it to the final set F, and we know all subsequent x-coordinates past its next intersection will have this line covered, so we remove it and move to the next intersection point. The runtime of this algorithm is $O(n\log n)$ as it satisfies the equation $T(n) \leq 2T(n/2) + cn$, where we recursively divide L in half and there are 2 traversals outside of the recursions so $c = 2$, so by Master's Theorem, we have $O(n\log n)$.