CS180, Winter 2018
Homework 6
Problems: 6.19,5.2,5.3,5.5

Derek Xu

February 22, 2018

1 Problem 6.19

First, we define the following problem:
Let $x = (x_0, x_1, ..., x_{n_x})$ be some string of 1s and 0s of length n_x
Let $y = (y_0, y_1, ..., y_{n_y})$ be some other string of 1s and 0s of length n_y
Let x' be a string equal to the substring x repeated
Let y' be a string equal to the substring y repeated
Let S' be the set of all strings such that it is equal to x' with elements from y' inserted at arbitrary locations, in order.
We must find if a string $s = (s_0, s_1, ..., s_n)$ exists in S'.
Note: n is the length of s

We take a dynamic programming approach to solving this problem.
We attempt to reduce the problem by assuming an optimum solution for a substring of s. In this case, we observe the substring $s - s_0$ where s_0 is the first character in the the substring s. We notice that s exist in S' if s_0 is either the first character of x_0 or the first character of y_0. We notice that if such is the case, then s_1, the next character of s, must be either x_1 or y_1. We notice that a special case arises at s_{n_x}, assuming that $(s_1, s_2, ..., s_{n_x}) = (x_1, x_2, ..., x_{n_x})$. Here, the next character of s must be either y_i or x_0, where i is an arbitrary value from previous iterations, otherwise x_i would not repeat a proper number of periods. Hence the following relationship is formed:

```
1 if $s_0 = x_i$:
2         $option_a = OPT(s_1, ..., s_n, x_{i+1}, y_i)$
3 else if $s_0 = y_i$:
4         $option_b = OPT(s_1, ..., s_n, x_i, y_{i+1})$
5 else:
6         $OPT(s_0, s_1, ..., s_n, x_i, y_i)$ is false.
7 if $option_a$ or $option_b$ is true:
8         $OPT(s_0, s_1, ..., s_n, x_i, y_i)$ is true.
```

Note: Here we define that $x_{n_x+1} = x_0$ and $y_{n_y+1} = y_0$.
...which is a O(1) operation

We now look at the base case:
For $OPT(s_n, x_i, y_i)$, we simply check if $(x_i = x_{n_x}$ and $y_i = y_0)$ or $(y_i = y_{n_y}$ and $x_i = x_0)$. If either is the case, then the sequence belongs to S'. Otherwise, it does not belong to S'.

Finally we put this into a dynamic programming approach:
1 Let x and y as globals (for base case).
2 Initialize $OPT(s, x_i, y_i)$ as an array.
4 Return $OPT(s, x_0, y_0) \Rightarrow$ recursive loop mentioned above.

Note, some optimizations can be made here. Instead of passing s to OPT(), we notice that we can pass a pointer to the first element of s, s_0, as that is the only element we look at for that recursion. In this case, the base case would be when s is at index $n+1$, in which case, we simply see if the $x_i = x_0$ and $y_i = y_0$ as mentioned above. This means that our algorithm returns false only when the sequence cannot be covered by COMPLETE repetitions of x and y, which is what we want (proof of Correctness). Instead of computing recursively, we can also compute this bottom up with the base cases first, which saves call stack space complexity. In this case we would start at $OPT(n, x_i, y_i)$, as our answer will be returned at $OPT(1, x_i, y_i)$, because our first parameter i iterates from 1 through n. Note, with these optimizations we achieve the following: the recursive calls are all O(1) complexity, and we have n values for s and most likely values for x_i and y_i, $n_x << n$ and $n_y << n$, computing all values in the table will take roughly $O(n)$ time. If the above 2 inequalities are not the case, our algorithm runs in time $O(nn_xn_y)$. we notice that the maximum value for n_x and n_y is $n/2$, thus in this case our algorithm is O(n^3), but for such a case, we would be more inclined to use other algorithms such as maximum common subset. This case does not exploit the repetition either. In the case where $n_x << n$ and $n_y << n$, our algorithm runs in linear time O(n). I checked with a TA, and this discrepancy classifies as an "efficient algorithm", as the problem dictates.

2 Problem 5.2

First, we define the following problem:
Let A be a sequence of n numbers $a_1, a_2, ..., a_j$.
Let an inversion be the situation where $a_i < a_j$ but $i > j$.
Let a significant inversion be the situation where $a_i < 2a_j$ but $i > j$.
We must find the number of significant inversions.

We notice that finding the number of significant inversions is equivalent to finding the number of inversions except we do not count cases where $a_i < a_j$ but $i > j$ AND $a_i \geq 2a_j$ but $i > j$. Thus by running through the inversion algorithm and adding the extra step we get an algorithm of run time O(nlogn). The following pseudo code demonstrates this.

```
Sort-and-Count($L$)
    If the list has one elt
        there are no inversions
    Else
        Divide the list into 2 halves:
        ...$A$ contains the first $n/2$ elts
        ...$B$ contains the remaining $n/2$ elts
        $(r_A, A)$ = Sort-and-Count($A$)
        $(r_B, B)$ = Sort-and-Count($B$)
        $(r, L)$ = Merge-and-Count($A$,$B$)
Return ($r = r_A + r_B + r$, sorted list, $L$)

Merge-and-Count($A, B$)
    Maintain a $Cur$ pointer to each list initialized
    ...to point to the front elt's
    Maintain a $Cnt$ variable for the number of inversions
    ...initialized to 0
    While both lists are not empty:
        Let $a_i$ and $b_j$ be the elt's pointed by the $Cur$ pointer
        Append the smaller of these 2 to the output list
        If $b_j < 2a_i$
            increment $Cnt$ by number of elt's remaining in $A$
    Advance the $Cur$ pointer in the list from which the smaller elt
    ...was selected.
Return ($Cnt, Outputlist$)
```

Proof of Correctness (by contradiction):
Assume that we returned the wrong number of inversions. This means either our recursive step does not accurately compute the right number of inversions, or we computed the wrong number of inversions with the merging, or we computed the wrong number of inversions at the base case. Our base case is correct: if there are no elt's then there are no inversions. Our merging is correct. We count the number of time that $a_i \geq 2a_j$ but $i > j$, in the if statement on line 20. Our recursion is correct, this is because each time we invert, we get closer to the sorted solution. i.e. inverting in the subsequence will not introduce new inversions in the merged sequence. Because of this, our inversion for a sequence is the number of inversions for a set of disjoint subsequences that make up the the sequence and the number of inversions between the sorted disjoint subsequences. Therefore, by contradiction, our algorithm is correct.

Proof of Time Complexity:
This algorithm was proven to be nlogn as the merge step takes O(n) times and each recursion splits the problem into 2 subproblem half the input size. i.e. By Master's Theorem, our algorithm is T(n) = 2T(n/2) + n which is also O(nlogn).

3 Problem 5.3

First, we define the problem:
Let S be a set of credit cards $s_1, s_2, ..., s_n$
Let C be formed of disjoint subsets of credit cards, $C_1, C_2, ..., C_m$
Let $F(S)$ be a function that checks if s_i and s_j belong to the same subset
We must find if $\exists$ a subset C_i in C such that $|C_i| > n/2$

We notice that if given a set S, for S to return true, we can split S into 2 equally sized subsets S_1 and S_2, such that there must $\exists$ a $C_j \epsilon S_i$ where $C_j > n/4$. If such is not the case, when we merge the 2 subsets, the result cannot be $> n/2$, by the following inequality: $a_1 < b, a_2 < b$ implies $a_1 + a_2 < 2b$. In other words, the subset in S that is $n/2$ must also be $C_j \epsilon S_1$ or $C_k \epsilon S_2$ such that $C_j, C_k > n/4$. Note, we only need to reach time complexity O(nlogn). By dividing the problem into 2 subproblem that is 1/2 the size of the original problem, we can observe by Master's theorem that our merge can take time O(n).

A simple algorithm for this is as follows:

```
1 Find-subset ($S$):
2       Let $C_k$ be null.
3       Split $S$ into 2 sets $S_1$ and $S_2$ such that $|S_1|, |S_2| \leq \lceil n/2 \rceil$
4       Let $C_i$ be Find-subset($S_1$)
5       Let $C_j$ be Find-subset($S_2$)
6       If $C_i$ is not null:
7           Let $C_k$ be $C_i$
8           For $s_i$ in $S_2$
9               If $F(s_i, C_i)$
10                  Add $s_i$ to $C_k$
11      If $|C_k| > n/2$, Return $C_k$
12      If $C_j$ is not null:
13          Let $C_k$ be $C_j$
14          For $s_i$ in $S_1$
15              If $F(s_i, C_i)$
16                  Add $s_i$ to $C_k$
17      If $|C_k| > n/2$, Return $C_k$
18      Return null.
```

Proof of Correctness (by contradiction):
Assume that we returned the wrong answer. We either returned a set C_k that is not $|C_k| > n/2$ or we did not return a subset when a subset exists. The former is false as we check for that case before returning on lines 11 and 17. Now for the latter: We have shown that for a favorable C_k to exist, that a subset of C_k must appear from the 2 subproblems. Thus, the latter case would imply an error in the merging of the 2 subsets. But, for this we simply linear scan through all possible points that may belong to one of the subsets returned from the 2 subproblems (this is simply the points from the other subproblem as we already know that all points that are not in the subset in the first subproblem... is not in the subset). Thus, we're running a brute-force-ish way of merging which is correct, trivially. Therefore, by contradiction, our algorithm is correct.

Proof of Time Complexity:
As mentioned above our merging section takes O(n) time as we are traversing at most each node in each subgraph once. Because we split our problem into 2 subproblems of size n/2, we get the following recurrence relation: T(n) = 2T(n/2) + O(n). By Master's Theorem, this is O(nlogn).

4 Problem 5.5

First, we define the problem:
Let L be a set of lines $L_1, ..., L_n$
Let each line, L_i, consist of a slope and an intercept: a_i, b_i
In this case we can find the y value with $y = a_i x + b_i$.
Let a line, L_i, be uppermost if it is not below the set of lines $L - L_i \forall x \epsilon \mathbb{R}$.
We must find the maximum set of uppermost lines.

We notice that for the specified time O(nlogn), we will probably use a divide and conquer algorithm. We assume that we can solve for 2 subsets of the set and get their uppermost lines. We must now merge the 2 subsets to get the uppermost lines of the larger set in time O(n)... from Master Theorem as described in the previous 2 problems. To do this let's first observe some base and special cases.
For merging 0 lines, the uppermost line is nothing
For merging 1 line, the uppermost line is simply that line
For merging 2 lines, the uppermost line is both, unless the 2 lines are parallel in which case the uppermost line is the line with the higher intercept
For merging 3 lines, its either all 3 lines are parallel which results in the same answer at 2; 2 lines are parallel which we remove one and keep the other 2; or 3 lines are all not parallel which we find the intercept between 2 lines with highest and lowest slope and then see in the 3rd line is above or below the intercept. Note, the intercept denotes the smallest value on the 2 lines.

For merging n lines, we may be tempted to use a similar algorithm as 3 lines, but we notice the following case: say that we have found that the 1st to ith highest and lowest slope lines are all uppermost. As with the algorithm for the 3rd line, we need to check the (i+1)th line, by seeing if it is below or above the intercept. This is problematic because the line can cover anywhere between no lines and all lines, meaning we must check all intercepts, which is of time O(i). But, we have n iterations of i, starting from i=1. Thus we get a run time of O(1+2+...+n)=O(n^2). We must then use some other algorithm.

We notice that the key idea is comparing the new lines to the intercept points of the previous lines. We notice that the highest and lowest slope lines must be part of the solution. We notice that we can assume that all the lines from both subsets are uppermost. Notice, the highest and lowest slope values have importance. Thus, we may gain new insight if we sort our lines by slope. This can be run in time O(nlogn) with merge sort. Thus, it is valid to use. Now we can assume all of the uppermost lines from subset 1 strictly higher slope than the uppermost lines from subset 2.

We now notice that another key point that is missing from the O(nlogn) solution: there will be, in the best case, no lines that block other lines in which case we return the uppermost lines of both set. Conversely, in the general case we can find 1 line in either subset that will cover many lines from the other, such that all other lines in that subset covers the same or less lines than that line. This lines is the smallest slope out of the Our problem now is to determine what line this is in O(n) time.

Some methods that do not work: This line is not the smallest/biggest of the bigger/smaller slope subset. This line can come from either subset. This line cannot be compared to all intersections in other set (in O(nlogn)). i.e. we must compare line with only 1 intersection. but line can cover any number from 1 to m other lines. Thus the answer probably isn't in adding lines.

We may be tempted to look purely at the intersections, but this is incorrect as it fails to capture the lines at other points if the 2 subgraph intersections are far apart. But, we realize that intersections are the critical points in which we need to determine the uppermost lines. Here, we find a solution. We can return the intersections in the subproblems, merge the intersections, evaluate for uppermost line at each intersection, remove and add lines accordingly, and the return the new set of uppermost lines and intersections. For the base case, again the above algorithm describes merging 0, 1, 2, and 3 lines.

We now consider the algorithm for n lines:

```
1  Find-uppermost($G$)
2      Let $I_k, L_k$ be null.
3      Split $G$ into 2 sets $G_1$ and $G_2$ such that $|G_1|, |G_2| \leq \lceil n/2 \rceil$
4      Let $(I_1, L_1)$ be Find-uppermost($G_1$)
5      ...where $I_1$ is the set of intersections, and $L_1$ the set of
6          ...corresponding uppermost lines.
7      ...Note: for $i_j \epsilon I_1$, $l_j \epsilon L_1$ is the line to the left of $i_j$ and $l_{j+1} \epsilon L_2$ is
8          ...line to the right
9      Let $(I_2, L_2)$ be Find-uppermost($G_2$)
10     Let $I_{ms}, L_{ms}$ be merge($I_1 \cup I_2$) such that $L_1$ and $L_2$ are swapped
11     ...in correct order as this merge is from merge sort
12     Add $L_{ms}(0)$ to $L_k$
13     For all values $i_o$ in $I_{ms}$:
14         Find the highest uppermost line at point $i_o$, $l_k$
15         ...can be found at O(1) by examining the lines connected to
16             or closest to from the right to this intersection
17         If $l_k(i_o)$ is greater than the $y$ value of $i_o$
18             find the intersection of $l_k$ with $L_k(o)$, call it $i_n$.
19             ...Note, we can skip to the next intersection of $l_k$ here.
20         add either $i_o$ or $i_n$ to $I_k$ (depending on line 18) and $l_k$ to $L_k$
21     Return $I_k, L_k$
```

Proof of Correctness (by contradiction):
Assume that we returned the wrong answer. This must mean our output is not the uppermost lines, which can be at fault either in our base case or when we merge the subproblems. We note that our base cases are correct, which proof's trivial. Thus, the fault must be in our merging. Notice, in are merging routine we find the uppermost line at each intersection. Thus, our assumption is that the uppermost lines at each intersection constitutes a set of uppermost lines for the entire set. This is true: observe the order by which we are determining the uppermost line, and the order by which we are adding lines to the set. By evaluating the uppermost line, we are determining either the line in the set with lower slopes or a line in the set with higher slopes. If the lower slope line is uppermost, we simply can continue to the next intersection. Any other intersection between the current and next intersection will be due to the uppermost line of the other intersection being

of higher slope, which we account for in line 18 of our algorithm. If the higher slope line is uppermost, we can conclude that the lines after that uppermost line will also be higher slope. This is because the value at points of x greater than the intersection of the lower slope line grows slower than the value at points of x greater than the intersection of the higher slope line, by basic geometry. Thus we may again move to the next intersection and evaluate. Thus this assumption is true. Therefore, by contradiction, our algorithm is correct.

Proof of time complexity:

We have shown that by Master's Theorem (see prev. questions), all we need to show is that the merging step is O(n). We do this by going through the code. we first merge the 2 result entries. The time for merge is O(n). Then, we go through all of the intersections, which we proved is less than the number of lines making it O(n). Then we run an O(1) find k_j and O(1) operations. The total run time is O(n) + O(n) $\cdot$ O(1) = O(n). Thus the total algorithm (again by Master's Theorem) runs in time O(nlogn).