Computer Science 180, Homework 6

Michael Wu
UID: 404751542

February 22nd, 2018

Chapter 6, Problem 19

First we set $x' = x^k$ for some k such that x' is at least the length of s. Similarly for y let $y' = y^k$ for some k such that y' is at least the length of s. If s is an interleaving of x and y, denote s' to be the subsequence of s that is a repetition of x and denote s'' to be the subsequence of s that is a repetition of y. To solve this problem, we consider the subproblems $\text{OPT}(i, j)$ where

$$\text{OPT}(i,j) = \begin{cases} \text{true} & \text{if } \exists s' \text{ s.t. } |s'| = i \text{ and } \exists s'' \text{ s.t. } |s''| = j \\ \text{false} & \text{otherwise} \end{cases}$$

for a string t of length $|t| = i + j$. Our base case is $\text{OPT}(0, 0) = \text{true}$.

Then for our given string s, we consider the substring s_1 containing only the first character . This is an interleaving of x and y if and only if $\text{OPT}(1, 0) = \text{true}$ or $\text{OPT}(0, 1) = \text{true}$. Let us access the ith character of a string a using the zero indexed notation $a[i - 1]$. $\text{OPT}(1, 0)$ and $\text{OPT}(0, 1)$ are true only if $s[0] = x'[0]$ or $s[0] = y'[0]$, respectively. This tells us whether s_1 is an interleaving of x and y.

Then we can consider the substring s_2 containing the first and second characters, and calculate $\text{OPT}(2, 0)$, $\text{OPT}(1, 1)$, and $\text{OPT}(0, 2)$ using the following recurrence

$$\text{OPT}(i,j) = \begin{cases} \text{true} & \text{if } \text{OPT}(i-1,j) = \text{true and } x'[i-1] = s[i+j-1] \\ \text{true} & \text{if } \text{OPT}(i,j-1) = \text{true and } y'[j-1] = s[i+j-1] \\ \text{false} & \text{otherwise} \end{cases}$$

which corresponds to assigning the next character in our string to either x' or y'. We can continue to use this recurrence, each time considering a string that increases in length by 1, until we calculate every possible OPT(i, j) such that $i+j$ is equal to the length of our original string s. If such an OPT$(i, j) =$ true, then it is true that s is an interleaving of x and y. Otherwise s cannot be an interleaving of x and y. In full our algorithm is as follows.

```
boolean checkInterleaving(String s, String x, String y){
   while(x.size()<s.size())
      x=x.concat(x);
   while(y.size()<s.size())
      y=y.concat(y);
   boolean[][] opt=new boolean[s.size()+1][s.size()1+1];
   opt[0][0]=true;
   for(int i=1;i<=s.size();i++)
      for(int j=0;j<=i;j++){
         opt[i-j][j]=false;
         if(i-j-1>0 && opt[i-j-1][j]==true
         && x.charAt(i-j-1)==s.charAt(i-1))
            opt[i-j][j]=true;
         if(j-1>0 && opt[i-j][j-1]==true
         && y.charAt(j-1)==s.charAt(i-1))
            opt[i-j][j]=true;
         if(i==s.size() && opt[i-j][j]==true)
            return true;
      }
   return false;
}
```

This has a runtime of at most $O(s^2)$, because we calculate a 2-D array that has a height and width that grows linearly with s. Its correctness is a result of the recurrence relation given above.

Chapter 5, Problem 2

We simply modify the inversion algorithm by maintaining another pointer p that gives us how many significant inversions there are when merging. More explicitly, when we place an element r from the right half R before the ith element of the left half L, we set $p = i$ and increment it by one until we get the $p = j$th element l_j such that $l_j > 2r$. Then we know there are $|L| - j + 1$ significant inversions caused by this placement of r because L is sorted. For subsequent placements of elements $r \in R$, we only change p by incrementing it or moving forward to match the first unplaced element in L. Thus p traverses L once, which takes linear time. So our merge function still takes linear time, leading to an overall $O(n \log n)$ runtime for our inversion algorithm. In full this algorithm is as follows.

```
int mergeCountInv(List<Integer> S,int start,int end) {
   if (end-start<=1)
      return 0;
   int mid=(start+end)/2;
   int inversions=0;
   inversions+=mergeCountInv(S,start,mid);
   inversions+=mergeCountInv(S,mid,end);
   inversions+=merge(S,start,mid,end);
   return inversions;
}
int merge(List<Integer> S,int start,int mid,int end) {
   List<Integer> merged=new ArrayList<Integer>();
   int inversions=0;
   int indL=start;
   int indR=mid;
   int indInv=start;
   while (indL<mid || indR<end)
      if (indL==mid)
         merged.add(S.get(indR++));
      else if (indR==end)
         merged.add(S.get(indL++));
      else if (S.get(indL)<=S.get(indR)) {
         merged.add(S.get(indL++));
      else {
```

```
            merged.add(S.get(indR++));
            if(indInv<indL)
               indInv=indL;
            while(S.get(indInv)<=2*S.get(indR) && indInv<mid)
               indInv++;
            inversions+=mid-indInv;
         }
   for (int i=0;i<merged.size();i++)
      S.set(start+i, merged.get(i));
   return inversions;
}
```

Here we merge sort our list S in place, and use the integer `indInv` as a pointer to track how many significant inversions there are when merging.

Chapter 5, Problem 3

We divide up the set S of n cards into two piles of at most $\frac{n}{2}$ cards. Continue to do this recursively, returning piles of size 1. We merge piles by noting that if both the left L and right R piles of cards do not have a set where more than half of the cards in the pile are all equivalent to one another, then $L \cup R$ cannot have a set where more than half the cards in $L \cup R$ are all equivalent to one another.

If L has a set S_l where more than half the cards in L are equivalent, we need to check every element $e \in R$ to see if e is equivalent to $x \in S_l$. Let S_{match} be the set of $e \in R$ that are equivalent to $x \in S_l$. Then if $|S_l \cup S_{match}| > \frac{|L \cup R|}{2}$, we know that a set exists such that more than half the cards in $L \cup R$ are all equivalent to one another.

Similarly for R, if R has a set S_r where more than half the cards in R are equivalent, we need to check every element $e \in L$ to see if e is equivalent to $x \in S_r$. Let S_{match} be the set of $e \in L$ that are equivalent to $x \in S_r$. Then if $|S_r \cup S_{match}| > \frac{|L \cup R|}{2}$, we know that a set exists such that more than half the cards in $L \cup R$ are all equivalent to one another.

At most we need to check every element in L and R once, so this merge step will take linear time in $|L \cup R|$. After each merge return the equivalence set that has a size of at least half of the merged set $L \cup R$, if it exists. We merge all the way up until we get a result for our original set S. In full the algorithm is as follows.

```
(Card, int) checkFraud(List<Card> S) {
   if(S.length==1)
      return (S.get(0),1);
   List<Card> left=S.subList(0,S.size()/2);
   List<Card> right=S.subList(S.size()/2,S.size());
   Card leftCard, int leftSize=checkFraud(left);
   Card rightCard, int rightSize=checkFraud(right);
   if(leftCard==null && rightCard==null)
      return (null,0);
   int size=0;
   Card card=null;
   if(rightCard!=null) {
      int matches=0;
      for(int i=0;i<left.size();i++)
```

```
            if(sameAccount(rightCard,left.get(i)))
               matches++;
         if(matches+rightSize>S.size()/2) {
            size=matches+rightSize;
            card=rightCard;
         }
      }
      if(leftCard!=null) {
         int matches=0;
         for(int i=0;i<right.size();i++)
            if(sameAccount(rightCard,left.get(i)))
               matches++;
         if(matches+leftSize>S.size()/2) {
            size=matches+leftSize;
            card=leftCard;
         }
      }
      return (card, size)
   }
```

This code mostly uses Java syntax, except for when it returns a tuple. When there exists a set X of more than half of the cards in S that are all equivalent to one another, this function returns the size of this set $|X|$ and a card $x \in X$ from this set. Otherwise it returns null. It suffices to keep track of only these because all we need during the merge is x to check for equivalence and $|X|$ to determine if there are enough matches m such that $|X| + m > \frac{|S|}{2}$.

The correctness of this algorithm comes from the previous argument. It has a runtime $\mathrm{T}(n)$ equal to

$$\mathrm{T}(n) = 2\,\mathrm{T}\left(\frac{n}{2}\right) + O(n)$$

which is the same as merge sort. Thus it takes $O(n \log n)$ time.

Chapter 5, Problem 5

We can actually find the visible lines in linear time if we first sort the lines by increasing slope m. Using merge sort takes $O(n \log n)$ time, which determines the time complexity of our algorithm. Denote a line as $l = (m, b)$ where m is the slope and b is the intercept.

After sorting, we traverse our set S of lines in linear time and remove any parallel lines that are blocked. This is easy to do because parallel lines will be adjacent to each other in S, since S is sorted by slope. We simply remove the line with a smaller b when we encounter two parallel lines next to each other in S. If two lines have the same slope m and intercept b, remove either because they are the same line.

Now S contains lines sorted by increasing slope, and no two lines in S have the same slope. To figure out which lines are visible, we add the first two elements in S to our set of visible lines V. If $|S| \leq 2$, we can simply return every line in S, because they all must be visible. This is because every line in S must intersect, as none are parallel. For two lines, there exists some intersection point p between the first line l_1 and the second l_2 such that l_1 is visible on the interval $(-\infty, p)$ and the l_2 is visible on the interval (p, ∞).

Then we consider the third line $l_3 \in S$ and add it to V. This line must be visible as we move towards ∞, since it has the greatest slope in V. Next we must determine if it blocks l_2. This happens if the x value of the intercept between $l_1 = (m_1, b_1)$ and $l_3 = (m_3, b_3)$ is less than the x value of the intercept between l_1 and $l_2 = (m_2, b_2)$. In other words, we remove l_2 from V if

$$\frac{b_3 - b_1}{m_3 - m_1} < \frac{b_2 - b_1}{m_2 - m_1}$$

This is true due to the fact that $m_3 > m_2$. Note that we do not need to consider the case where l_1, l_2, and l_3 intersect at the same location, as we assume no three lines intersect at the same location.

We continue adding each line $l_i \in S$ as $l_j \in V$, checking to see if $l_j \in V$ blocks l_{j-1} by looking at the intercepts of l_j with l_{j-2} and l_{j-1} with l_{j-2}. If a blocking occurs, we remove l_{j-1} from V and check again if l_j blocks the new l_{j-1}, removing each blocked line going backwards in V until either $|V| = 2$ or l_{j-1} is not blocked. If l_{j-1} is not blocked, then every line previous to it must not be blocked either by l_j and we can stop there. This is because previously l_{j-1} was visible on an interval (p_{j-1}, ∞), but with the addition of

l_j it is visible on the interval (p_{j-1}, p_j). Thus l_j is visible on (p_j, ∞), and cannot block any lines visible on the interval $(-\infty, p_{j-1})$ since $p_{j-1} \leq p_j$.

Although this process may seem to have quadratic time complexity, as we need to look backwards in V and remove blocked lines for every line in S, it actually is linear. Every line in S can only be added once to V and removed once from V, leading to $2|S| = 2n$ steps. Thus our entire algorithm scales with the time it takes to merge sort S, and takes $O(n \log n)$ time. In full our algorithm is as follows.

```
List<Line> getVisibleLines(List<Line> lines) {
   lines=mergeSortBySlope(lines);
   int i=0;
   while(i<lines.size()-1)
      if(lines.get(i).m()==lines.get(i+1).m())
         if(lines.get(i).b()>=lines.get(i+1).b())
            lines.remove(i+1);
         else
            lines.remove(i);
      else
         i++;
   if(lines.size()<=2)
      return lines;
   List<Line> visible = new ArrayList<Line>();
   visible.add(lines.get(0));
   visible.add(lines.get(1));
   for(i=2;i<lines.size();i++) {
      Line prev=visible.get(visible.size()-1);
      Line secondPrev=visible.get(visible.size()-2);
      visible.add(lines.get(i));
      double prevX = intercept(prev,secondPrev).getX();
      double currX = intercept(lines.get(i),secondPrev).getX();
      while(currX<prevX) {
         visible.remove(prev);
         if(visible.size()==2)
            break;
         prev = visible.get(visible.size()-2);
         secondPrev = visible.get(visible.size()-3);
         prevX = intercept(prev,secondPrev).getX();
```

```
            currX = intercept(lines.get(i),secondPrev).getX();
        }
    }
    return visible;
}
```