Cody Hubbard - 004843389

CS 180 HW6

CH5 #2

Solution:

Run a simple mergesort algorithm on the sequence, however set an additional condition where for every swap that mergesort would do, first it checks if $a_i > 2a_j$ and if so increments a "significant inversion" counter. Additionally, during the merge process set another conditional that checks if any element of the lesser subsequence is greater than $2\cdot$(any element of greater sub array) [lesser and greater being determined by the index used in the original sequence]. Every time this conditional is found to be true increment the significant inversion counter as well.

Mergesort is known to run in time complexity$O(nlogn)$. The conditional will be checked some constant number of times C for each subsequence and the incrementation will happen some constant number of times K for each subsequence. These constants are insignificant to $O()$ notation and thus the complexity remains $O(nlogn)$.

This is correct becasue it recursively finds the number of significant inversions in the lesser subsequence, the greater subsequence, and then the merge process itself. This means that if that this method will work for any larger sequence the original sequence is a sub-problem of, as well as any subproblems spawned from the original sequence.

CH5 #3

Solution:

Do the following,

```
HalfEquiv (takes a $ParentSet$ of $setSize$ number of cards)
if $setSize = 1$: return the single card
if $setSize = 2$ : equivtest card1 & card2
  if equivtest is true: return card1
Break the current set of cards into two sets, $Cards1$ and $Cards2$
HalfEquiv($Cards1$)
  if this returns a card, $c$, use equivtest to test it against all other cards in $ParentSet$, if equivalences reach or become greater than $\frac{setSize}{2}$ return $c$
HalfEquiv($Cards2$)
  if this returns a card, $c'$, use equivtest to test it against all other cards in $ParentSet$, if equivalences reach or become greater than $\frac{setSize}{2}$ return $c'$
return no card exists
```

This algorithm is correct because when you divide the parent set into two conquerable halves one of those the two halves MUST have a majoroty card given such a card exists for the parent set. Additionally once the recusive algorithm is run on each subset atleast one of the calls will return a card, this card is then compared to all card is the parent set so that no matter what some majority card will be found adn this will be done recursively for all return majority card and all parent sets. Thus this algorithm will work for any problem that has our card set as a subproblem as well as any subproblem created from our inital set.

The number of times equivtest is called for any set of cards is *setSize*. Additionally for each recursive call it is run another $\frac{setSize}{2}$ times as well as a possible $2 \cdot setSize$ times for the testing of the parentset. Thus in total $2 \cdot (HalfEquiv(\frac{setSize}{2})) + 2 \cdot setSize = O(setSize \cdot log(setSize) = O(nlogn)$ as required.

CH5 #5

Solution:

Given a set of n lines specified by $y = a_i x + b_i$ for any $i - th$ line

Sort all lines by their slope, a_i

send the sorted lines to the following recursive function,

FindVis (takes a set of k lines)

if $k = 1$: return the single line

if $k = 2$:

consider the intersection of lines l_1 and l_2. If it exsits return the set of both lines, if it does no return the line whihc is uppermost (where uppermost is defined in the text as "f its y-coordinate at x_0 is greater than the y-coordinates of all the other lines at x_0: $a_i x_0 + b_i > a_j x_0 = b_j$ for all $j \neq i$")

split the lines into two sets, $L1 = l_1, ..., l_{\frac{n}{2}}$and $L2 = l_{\frac{n}{2}+1}, ..., l_n$

return Merging(FindVis($L1$) and FindVis($L2$))

wher Merging uses the following logic:

Since the slopes of lines in $L1$ are less than all the slopes in $L2$ we know that if any of the lines in $L1$ are visible their visible segment must be somewere left of the lines in $L2$, and we then know any pair of lines from $L1$ and $L2$ must only intersect at one point. Consider this intersection point of any such pair(l_i, l_j) where $l_i \in L1$ and $l_j \in L2$. If the x-coordinate of the intersection is less than the x-coordinate of all l_i's intersections all other lines from $L2$, remove the line from the set of visible lines.

The sorting step will take $O(nlogn)$ time and the merging step will take around $O(k)$ time where k is the size of each subproblem. therfore assuming we can find line intersections in time $O(nlogn)$ or less the total complexity for this solution will be soemthing like $2O(nlogn) + O(n) = O(nlogn)$ as required

This is correct because it checks visiblity during the smallest subproblem and also during merging.

CH6 #19

Solution:

given two repeating sequences X and Y are from the two ships, and S is the signal we're listening to which is of k digits

Let X'be a repetition of X up to n digits and Y'be a repetition of Y up to n digits.

Create a table $InlTbl[]$ to store the solutions of subproblems. The table will have a true value if a substring $S'$of S is found to be an interleave of X and Y.

Solving this problem then boils down to simply filling in the table.

Fill in the base cases of the table, $InlTbl[x, 0]$ and $InlTbl[0, y]$

```
  $InlTbl[x, 0] = TRUE$ if for $S(1 \rightarrow x) = X'(1 \rightarrow x)$ for $x \leq k$
  $InlTbl[0, y] = TRUE$ if for $S(1 \rightarrow y) = Y'(1 \rightarrow y)$ for $y \leq k$
```

Then all the other entries in the table can be found using the following

```
for $i \in 1 \rightarrow n$
  for $j \in 1 \rightarrow n$
    if $X'(i) = S(i + j)$ and $Y'(j) \neq S(i + j)$
      $InlTbl[i, j] = InlTbl[i - 1, j]$
    else if $Y'(j) = S(i + j)$ and $X'(i) \neq S(i + j)$
      $InlTbl[i, j] = InlTbl[i, j - 1]$
    else if $X'(i) = S(i + j)$ and $Y'(j) \neq S(i + j)$
      $InlTbl[i, j] = InlTbl[i - 1, j]$
    else if $X'(i) = Y'(j) = S(i + j)$
      $InlTbl[i, j] = InlTbl[i - 1, j]$
    else $InlTbl[i, j] = FALSE$
```

Then like siad, to solve the interleave problem you must simply check the table.

This algorithm is correct as long as the table is filled out correctly becasue each new sequence of additional length's solution is beased on the correctness of the previous entries in the table. That means that this method should give correct output for each subproblem of this sequence. It is also possible to extend this mehtod to any problem which has the given sequence as a subproblem.

The complexity of this method is $O(n^2)$ as the comparisons can be done in constant time, the accessing of elements in the table can be done in constant time, and the storing of boolean values in the table can be done in constatn time. This means that only the size of the loops matter which is $O(n^2)$.