CSE 417 HW 5 Problem 1

Yang Zhang 1030416

zhy9036@uw.edu

1. **Algorithm:**

```
EquivTester(set of cards S)
     n=|S|
     If n=1
         return the only card
     If n=2
         If card1 and card2 are equivalent
             return card1
     Let S1 be the first ⌊n/2⌋ cards.
     Let S2 be the remaining ⌊n/2⌋ cards.
     If EquivTester(S1) returns a card

         test the returned card against all other cards in S
         If you have not found the card for which more than n/2 cards
         are equivalent


         If EquivTester(S2) returns a card
              test the returned card against all other cards in S
         EndIf

      EndIf
      Return the card for which more than n/2 cards are equivalent
         if found

EndFunction
```

Proof:

This algorithm is correct because if more than n/2 cards are equivalent, then when it divide the whole set into two subsets, at least one of the half-sets will have more than half of the cards that equivalent to the whole set's majority equivalence. Therefore, one of the two recursive calls must return a card equivalent to the whole set's majority equivalence, and this algorithm compares all returned cards to the larger set, so the majority eequivalence will be found.

Recurrence Relation: **$T(n) = 2T(n/2) + 2n$, $a = 1$, $b = 2$, $c = 2$, $d = 1$, $k = 1$, therefore $T(n) = O(n \log n)$**

CSE 417 HW 5 Problem 2

Yang Zhang 1030416
zhy9036@uw.edu

2. Algorithm:

The main idea is to split each rectange shape into unit length rectange shapes, and then sorts the shapes by their first x postion. After that, devide the list of shapes into two sub-lists, until reaches the base case (list contains only 1 shape). During the merge step, compare the height if there are two shapes has the same x position and select the higher one into result list. Fininaly return the result list.

```
Initialization:

SortedList = sortByStartPoint(TupleList)
// split rectangle shape into unit length rectagnle
SortedList = split(SortedList);
function eliminationHidenLine2d(SortedList):
      if SortedList.size == 1:
            return SortedList;
      EndIf

      left_list = [left half of SortedList];
      right_list = [right half of SortedList];

      // Continune divding
      left_list = eliminationHidenLine2d(left_list);
      right_list = eliminationHidenLine2d(right_list);

      eliminated_left = [left_list[0]];
      for i in range(1, len(left_list)):
            if left_list[i] has the same x pos as the last element E in
                        eliminated_left
                  if left_list[i] has larger height, replace E with left_list[i]

            else eliminated_left.append(left_list[i]);
      EndFor

      eliminated_right = [right_list[0]];
      for i in range(1, len(right_list)):
            if right_list[i] has the same x pos as the last element E in
                        eliminated_left
                  if right_list[i] has larger height, replace E with right_list[i]

            else eliminated_right.append(right_list[i]);
      EndFor
```

```
      return [eliminated_left + eliminated_right];

EndFunction
```

Proof: Since in this problem shapes can only be covered by mutiple unit length, divided shapes into unit lengh would ensure that every conflicted position are not lost. And since this algorithm always select the higher shape when collision happened, therefore the correct silhouette will always be selected.

Recurence: **T(n) = 2T(n/2) + n = O(n logn)**

Non Divide-Conquer Solution:

```
Initialization:

SortedList = sortByStartPoint(TupleList)
// split rectangle shape into unit length rectagnle
SortedList = split(SortedList);
function eliminationHidenLine2d(SortedList):

      result = [Sortedlist[0]];
      for i in range(1, len(SortedList)):
            // check if colision exists
            if left_list[i] has the same x pos as the last element E in
                        SortedList
                  // select the higher one
                  if left_list[i] has larger height, replace E with left_list[i]

            else result.append(left_list[i]);
      EndFor
 EndFunction
```

The sorting step takes O(n logn) and the elimination step takes O(n), therefore the overall runtime is O(n logn)

CSE 417 HW 5 Problem 3

Yang Zhang 1030416
zhy9036@uw.edu

3. **(a)** exactly (n/2)* (n/2)* (n/2)* 8 = n^3 times
(b) T(n) = 8T(n/2) + Cn, a = 8, b = 2, k = 1
(c) Since, a > b^k, T(n) = O(n^($\log_b$a)) = O(n^3)
(d)

Level	Num	Size	Work
0	1=8^0	n	cn
1	8=8^1	n/2	8cn/4
2	64=8^2	n/4	64cn/16
…	…	…	…
i	8^i	n/(2^i)	(8^i)*cn/(2^i)
…	…	…	…
k	8^k	n/(2^k)	(8^k)*cn/(2^k)

T(n) = $\sum^{k}_{i=0}$ (8^i)*cn/(2^i) = cn * (4^{k+1} – 1)/(4 - 1) < cn * 4^{k+1} /3
= (4/3) * cn * (8^k / 2^k)
= (4/3) * cn * (8^($\log_2$n) / 2^($\log_2$n))
= (4/3) * cn * (8^($\log_2$n) / n)
= (4/3) * c * (n ^ ($\log_2$ 8))
= (4/3) * c * n^3 = O(n^3)

(e) T(n) = (n/2)* (n/2)* (n/2)* 7 = (7/8) * n^3 = O(n^3)
(f) T(n) = 7T(n/2) +cn, a = 7, b =2, k =1
since, a > b^k, T(n) = O(n^($\log_b$a)) = O(n^2.75)
(g) Strassen's method is **worse** than "obvious" algorithm in addition aspect, and it is **not depends on *n***. The reason is that those two method has same size deducation rate, in other word they have the same number of sub levels (regardless using recursion or not), while for each level, Strassen's method has more addtions than the "obvious" algorithm. Therefore, in total, Strassen's algorithm did more addtions than "obvious" algorithm.