CSE 417 HW5

Zachary McNulty (zmcnulty, 1636402)

February 2019

1 Problem 1

Q: Given algorithms with (exactly) the run times listed below, how much slower is each if (i) the input size is increased by 1, or (ii) input size is doubled?

a) $nlog_2(n)$

1. $$(n+1)log_2(n+1) = nlog_2(n+1) + log_2(n+1)$$
   As $nlog_2(n+1) \approx nlog_2(n)$, increasing the input size by one increases the runtime by around $log_2(n+1)$ units.
2. $$2nlog_2(2n) = 2n(log_2(2) + log_2(n)) = 2n + 2nlog_2(n)$$
   Thus, doubling the input size increases the run time by $2n + nlog_2(n)$ units.

b) n^2

1. $$(n+1)^2 = n^2 + 2n + 1$$
   . Thus, increasing the input size by 1 increases the run time by $2n + 1$ units.
2. $$(2n)^2 = 4n^2$$
   Thus, doubling the input size increases the runtime by a factor of 4 (i.e. increases the runtime by $3n^2$ units)

c) $100n^2$

1. $$100(n+1)^2 = 100(n^2+2n+1) = 100n^2+200n+100$$
   Thus, increasing the input size by 1 increases the runtime by $200n+100$ units.

2. $$100(2n)^2 = 4(100n^2)$$
   Thus, doubling the input size increases the runtime by a factor of 4 (i.e. increases the runtime by $300n^2$ units)

d) n^3

1. $$(n+1)^3 = n^3+3n^2+3n+1$$
   Thus, increasing the input size by 1 increases the runtime by $3n^2+3n+1$ units

2. $$(2n)^3 = 8n^3$$
   Thus, doubling the input size increases the runtime by a factor of 8 (i.e. increases the runtime by $7n^3$ units)

e) 2^n

1. $$2^{n+1} = 2*2^n$$
   Thus, increasing the input size by 1 doubles the run time (increases the runtime by 2^n units)

2. $$2^{2n} = 2^{n+n} = 2^n * 2^n$$
   Thus, doubling the input size increases the runtime by a factor of 2^n.

2 Problem 2

Algorithm

Run Breadth First Search, treating the n specimens as vertices and the m judgements as edges (i.e. a judgement between specimens i and j represents an edge (i, j) between vertices i and j on the graph. Mark the first vertex arbitrarily as A. Every time you explore an edge to an undiscovered vertex, if the edge is a "same" judgement, assign that vertex to the same group (A or B) as the current vertex, else if it is a "different" edge assign it to the opposite group as the current vertex. If you explore an edge going to a discovered vertex and would have assigned it a different group than it currently is set as, a consistent labeling is not possible. Else, if this never happens over the course of the algorithm, the labeling is consistent. If the graph happens to be disconnected, simply repeat the algorithm until all vertices have been discovered (i.e. once for each connected component).

DFS would also work fine.

Proof of Correctness
Our algorithm is sort of an extension of the test for Bipartiteness. In this case, we can see it would be impossible to label the specimens if there exists any cycle with an odd number of different edges. Each one of these edges MUST swap the group the vertex is placed in for the labeling to be consistent, so if there are an odd number in any cycle it would interfere with this: assign a specimen label A and by the end of the cycle you would want to assign it label B.

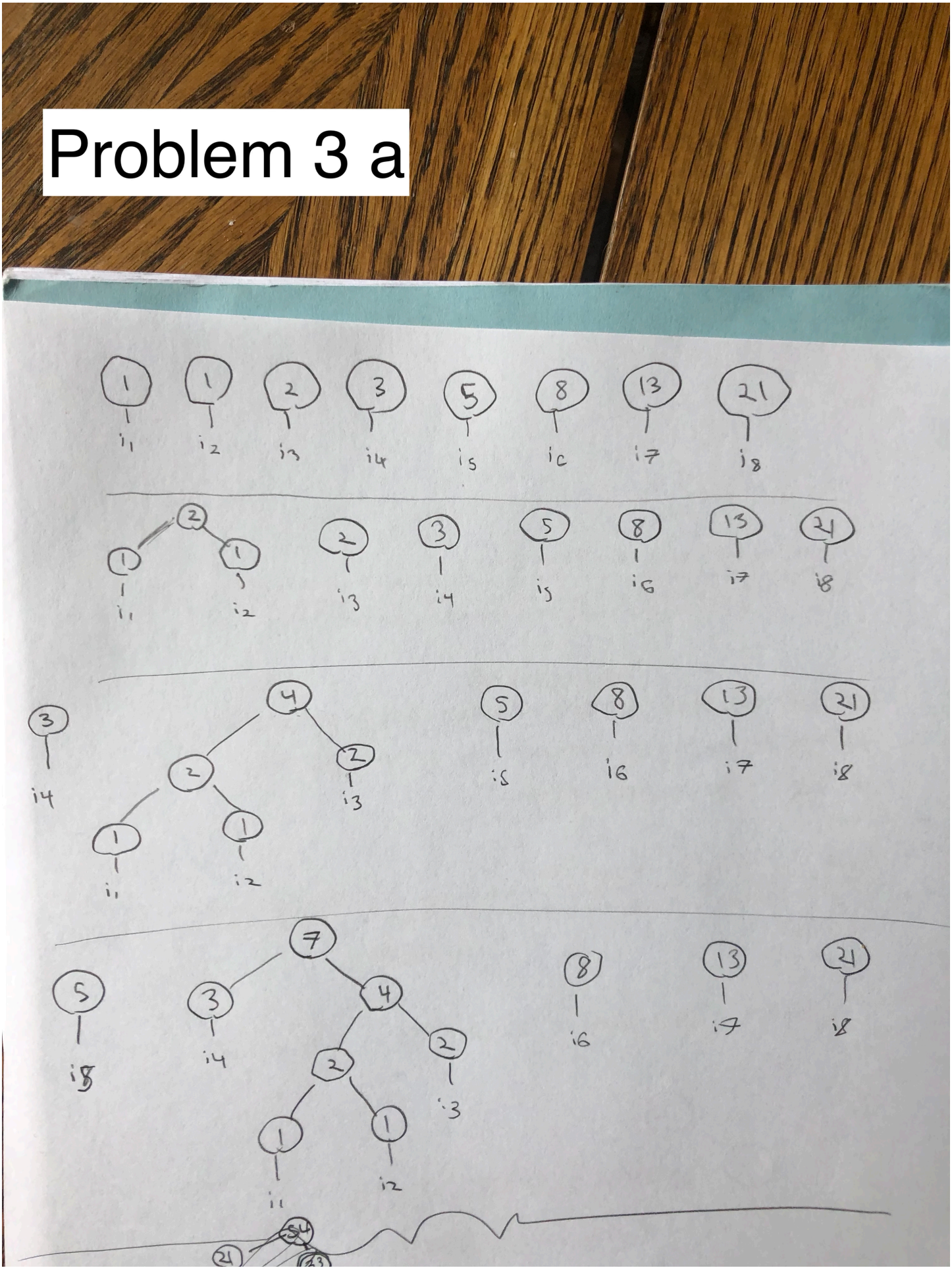
Suppose our algorithm reaches a point where it attempts to assign a label to a vertex v different from what that vertex was previously assigned while exploring an edge $e = (u, v)$ from vertex u. There are two possible scenarios. Both the vertices have the same current label yet the edge between them is "different", or both edges have different labels yet the edge between them is "same". In both cases, since both u and v have labels, there must be a path between them through the current BFS tree. Including the edge (u, v) with this path makes a cycle. In the first case, since the two have the same label, there is an even number of "different" edges along the path between u, v, so including the "different" edge (u, v) creates a cycle with an odd number of "different" edges. In the latter case, as the vertices were labeled differently there is an odd number of "different" edges between them, and adding the "same" edge (u, v) does not change that. In both cases, we find a cycle with an odd number of "different" edges, and thus a consistent labeling is not possible.

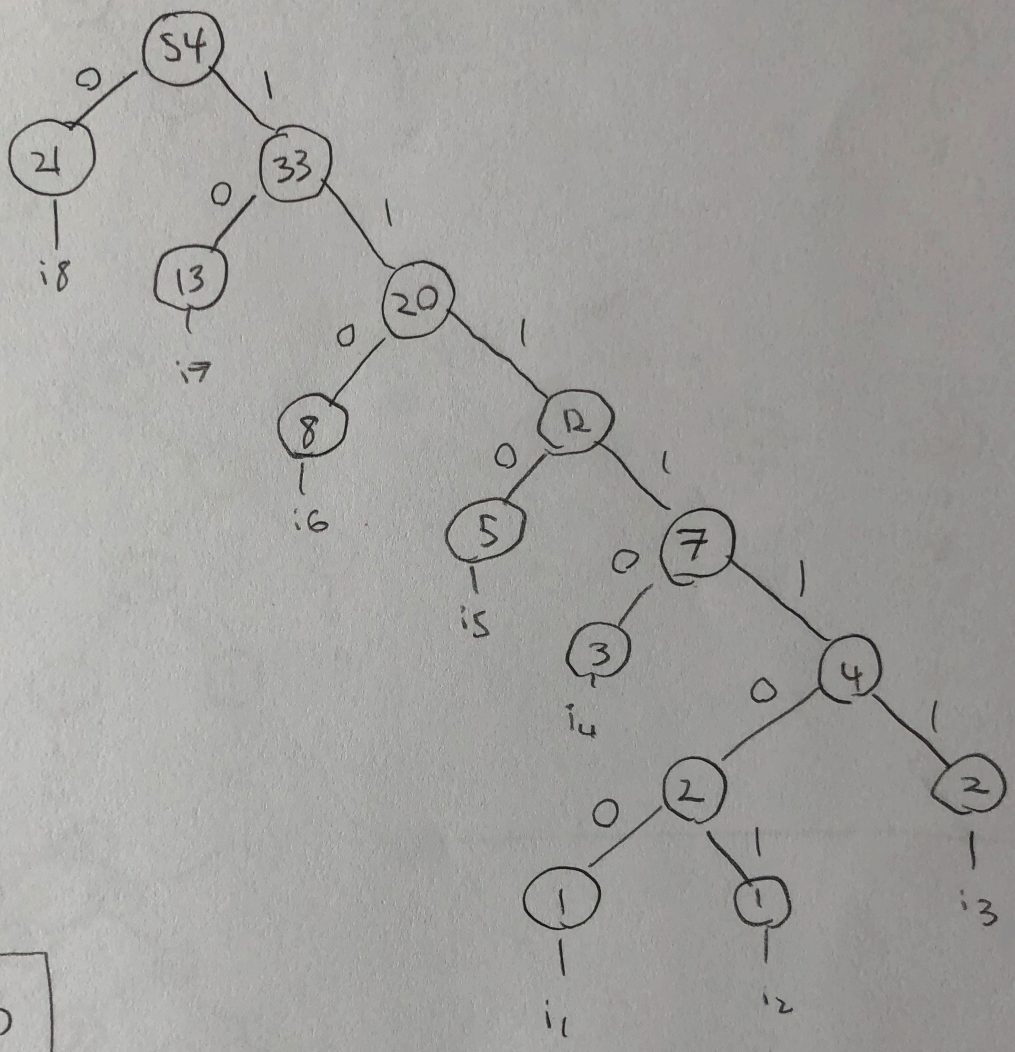
Complexity analysis

Generating an adjacency list is an $O(n + m)$ operations with n vertices and m edges, and BFS is also $O(n+m)$. Keeping track of what group each vertex is currently assigned to adds no more than constant time operations $O(1)$ to the

inner implementation of BFS, and thus does not change its complexity. Thus, overall the algorithm is $O(n + m)$

Problem 3 a


1 1 2 3 5 8 13 21
i1 i2 i3 i4 i5 i6 i7 i8
2
1 1
i1 i2
2 3 5 8 13 21
i3 i4 i5 i6 i7 i8
3
i4
4
2 2
1 1
i1 i2 i3
5 8 13 21
i5 i6 i7 i8
7
5 3 4 8 13 21
i5 i4 2 i6 i7 i8
2 i3
1 1
i1 i2
24
21 23



i_1 : 1 1 1 1 1 0 0

i_2 : 1 1 1 1 1 0 1

i_3 : 1 1 1 1 1 1

i_4 : 1 1 1 0

i_5 : 1 1 1 0

i_6 : 1 1 0

i_7 : 1 0

i_8 : 0

3 Problem 3

(a) The first 8 fibonacci numbers are $1, 1, 2, 3, 5, 8, 13, 21$ whose sum is 54. This gives the following character frequencies:

$$(i_1, i_2, i_3, i_4, i_5, i_6, i_7, i_8) = \frac{1}{54}, \frac{1}{54}, \frac{2}{54}, \frac{3}{54}, \frac{5}{54}, \frac{8}{54}, \frac{13}{54}, \frac{21}{54}$$

The Huffman Tree generated by these character frequencies is displayed above. When making the tree, we always placed the lower valued node to the left branch of the tree and the higher to the right. When nodes had equal value, we placed the node containing the lower indexed characters on the left. Zeros were assigned to left branches of the tree and ones to the right branches. The construction of the tree and the full tree are shown on the previous page.

(b) The average number of bits assigned by this code is:

$$7*\frac{1}{54}+7*\frac{1}{54}+6*\frac{2}{54}+5*\frac{3}{54}+4*\frac{5}{54}+3*\frac{8}{54}+2*\frac{13}{54}+\frac{21}{54} = \frac{22}{9} \approx 2.44$$

(c) In general, the Huffman code with n characters and character frequencies corresponding to the Fibonacci numbers, with the i^{th} character having frequency F_i, will have code:

$$i_1 = \{1\}^{n-3}00$$

$$i_2 = \{1\}^{n-3}01$$

$$i_3 = \{1\}^{n-2}$$

$$i_k = \{1\}^{n-k}0, \quad k > 3$$

where $\{1\}^n$ represents n ones. We get i_k by noticing the tree begins to simply branch very quickly to the left. This will always be the case because $F_n > \sum_{i=1}^{n-2} F_i$ for all $n \geq 3$ and because $F_n < \sum_{i=1}^{n-1} F_i$ for all $n \geq 3$ (proofs below) , and thus the lowest two nodes/frequencies to be combined in the Huffman Algorithm will always be the collection of nodes previously grouped into a tree and the lowest indexed i character (i.e. $(i_3, (i_2, i_1))$ and i_4 in the example above), with the lowest indexed i character to be the left term as it is the lower of the two nodes.

Proof: $F_n > \sum_{i=1}^{n-2} F_i$ for all $n \geq 3$

Base Case: $n = 3$

$$F_3 = 2 > 1 = F_1 = \sum_{i=1}^{1} F_i$$

Strong Induction

Assume $F_k > \sum_{i=1}^{k-2} F_i$ for all $3 \leq k \leq n$

$F_{n+1} = F_n + F_{n-1} \geq \sum_{i=1}^{n-2} F_i + F_{n-1} = \sum_{i=1}^{n-1} F_i$

Q.E.D.

Proof: $F_n < \sum_{i=1}^{n-1} F_i$ for all $n > 3$

Base Case: $n = 4$

$$F_4 = 3 < 4 = F_1 + F_2 + F_3$$

Strong Induction

Assume $F_k < \sum_{i=1}^{k-1} F_i$ for all $4 \leq k \leq n$

$$F_{n+1} = F_n + F_{n-1} < F_n + F_{n-1} + \sum_{i=1}^{n-2} F_i = \sum_{i=1}^{n} F_i$$

Q.E.D.

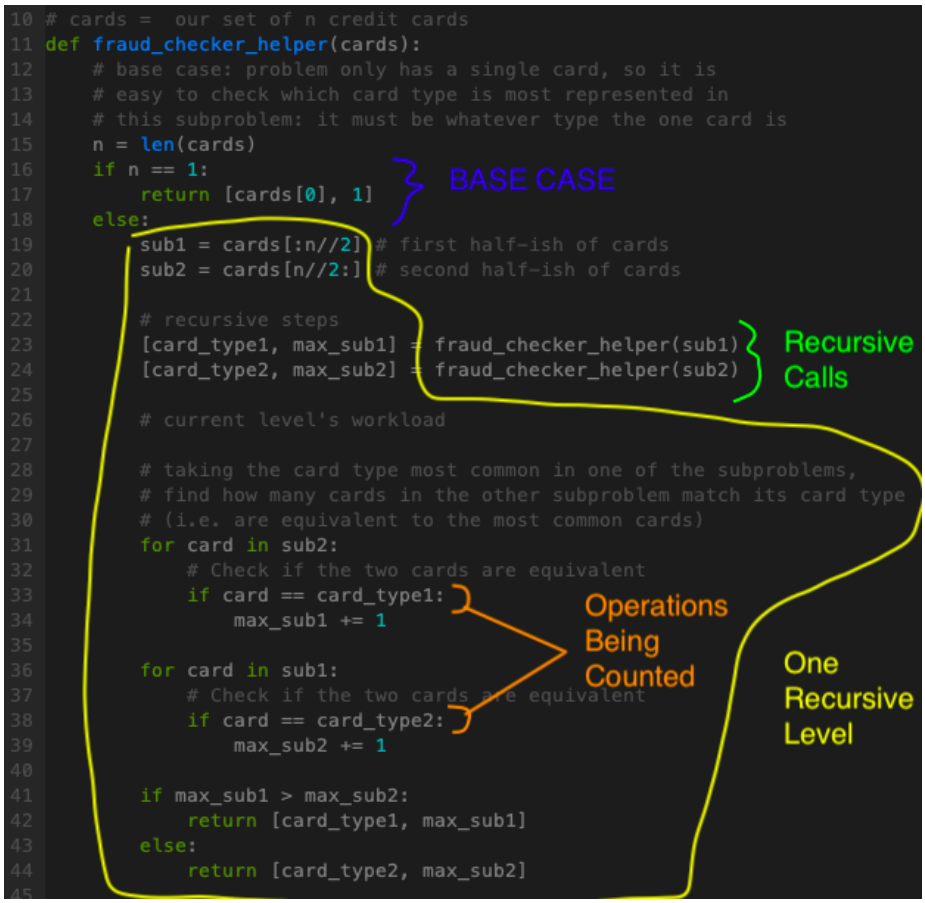
4 Problem 4

```python
# cards =  our set of n credit cards
def fraud_checker_helper(cards):
    # base case: problem only has a single card, so it is
    # easy to check which card type is most represented in
    # this subproblem: it must be whatever type the one card is
    n = len(cards)
    if n == 1:
        return [cards[0], 1]
    else:
        sub1 = cards[:n//2] # first half-ish of cards
        sub2 = cards[n//2:] # second half-ish of cards

        # recursive steps
        [card_type1, max_sub1] = fraud_checker_helper(sub1)
        [card_type2, max_sub2] = fraud_checker_helper(sub2)

        # current level's workload

        # taking the card type most common in one of the subproblems,
        # find how many cards in the other subproblem match its card type
        # (i.e. are equivalent to the most common cards)
        for card in sub2:
            # Check if the two cards are equivalent
            if card == card_type1:
                max_sub1 += 1

        for card in sub1:
            # Check if the two cards are equivalent
            if card == card_type2:
                max_sub2 += 1

        if max_sub1 > max_sub2:
            return [card_type1, max_sub1]
        else:
            return [card_type2, max_sub2]
```

Figure 1: Pseudocode for finding fraud!

Algorithm

Divide the cards in half and find the most common card (i.e. largest set of equivalent cards) in the left half and right half separately. To find these sets in each half, divide that half in half again and repeat the process. Continue dividing in half until we reach a set of one card where the answer is trivial. The most common card type is clearly that one card and the set of equivalent cards is clearly just one as it only contains that single card. Once you have computed each half's maximum equivalent card set, look for more equivalent cards in the other half. After doing this for both halfs, return the larger equivalent card set. If our max equivalent card set of the overall problem is greater than $n/2$, we have fraud!




```
# cards =  our set of n credit cards
def fraud_checker_helper(cards):
    # base case: problem only has a single card, so it is
    # easy to check which card type is most represented in
    # this subproblem: it must be whatever type the one card is
    n = len(cards)
    if n == 1:
        return [cards[0], 1]
    else:
        sub1 = cards[:n//2] # first half-ish of cards
        sub2 = cards[n//2:] # second half-ish of cards

        # recursive steps
        [card_type1, max_sub1] = fraud_checker_helper(sub1)
        [card_type2, max_sub2] = fraud_checker_helper(sub2)

        # current level's workload

        # taking the card type most common in one of the subproblems,
        # find how many cards in the other subproblem match its card type
        # (i.e. are equivalent to the most common cards)
        for card in sub2:
            # Check if the two cards are equivalent
            if card == card_type1:
                max_sub1 += 1

        for card in sub1:
            # Check if the two cards are equivalent
            if card == card_type2:
                max_sub2 += 1

        if max_sub1 > max_sub2:
            return [card_type1, max_sub1]
        else:
            return [card_type2, max_sub2]
```

Figure 2: Pseudocode marked up to reveal elements of recurrence

Correctness

If the max card set, of card type x, has size greater than $n/2$, we know that one of the halves we divide the cards into MUST have over $n/4$ equivalent cards in it. If both halves had less than or equal to $n/4$ of the equivalent cards in it, then the total number of equivalent cards would be less than or equal to $n/4 + n/4 = n/2$ a contradiction. Following the same logic, this continues to hold for further layers of recursion. In the half with more than $n/4$ equivalent cards, one of its subproblems will have more than $n/8$ equivalent cards. Thus, at each stage one of the subproblems at that level will be dominated by card x, so it will always be returned as one of the max card sets. Thus, at the initial level it will be returned as the max card set.

Complexity analysis

Define $C(n)$ to be the number of card equivalence comparisons. If we are at our base case, where there is only one card, we simply return that card in a single operation. Else, we can see that the set of n cards gets divided into two subproblems of approximately size $\frac{n}{2}$. Thus, at each stage we have $2C\left(\frac{n}{2}\right)$ recursive work. From this recursion, we get two cards which are equivalent to the greatest number of cards possible in their subproblem. With each of these cards, we check if they are equivalent to any of the other cards in the OTHER subproblem. Since every card undergoes exactly one comparison in this part, there are n comparisons (card equivalence checks) in total. Thus:

$$C(n) = \begin{cases} 1 & n == 1 \\ 2C\left(\frac{n}{2}\right) + n & n > 1 \end{cases}$$

Using the Master Recurrence Theorem, we can see that $a = 2, b = 2, c = 1, d = 1, k = 1$ so this suggests $C(n) \in O(nlog(n))$ as $2 = 2^1 \rightarrow a = b^k$

5 Problem 5

Algorithm

This algorithm breaks down the problem into a series of subproblems. By splitting the list of tuples in half, if I can find what the silhouette of each half separately (i.e. two sets of pairs L $\{(x_i, y_i)\}$ and $R = \{(x_j, y_j)\}$ both sorted by ascending x), I can get the silhouette formed when combining the two by following the following algorithm:

Run a modified version of the merge algorithm used during merge sort to join these two lists of pairs.

1. In the base case, finding the silhouette is easy: just use the corner points of the rectangle, (x_L, h) and $(x_R, 0)$.

2. Given two silhouettes, L and R, sorted by x coordinate as calculated in the subproblems, we can merge them into a new silhouette. Consider points in order of their x coordinate, running the merge part of merge sort. Keep track of the current heights of each of the two silhouettes described by L and R as yL and yR respectively. For example, if the last point I considered was $(x_i, 6)$ from L, I would set $yL = 6$ for the next point to be considered. When we consider the next point, there are three possible outcomes.

   (a) The new point's height is above the height of the opposite group's current height. For example, maybe $yL = 5$ and the next point I choose is $(x_i, 7) \in R$. In this case, add this point to the combined silhouette.

   (b) The new point's height is below the opposite group's current height, but its group's current height is still greater. For example, maybe $yR = 7, yL = 5$ and the next point is $(x_i, 3) \in R$. Even though $3 < yL$ we also have $yR > yL$ so we can still see the line going from the R silhouette heading down to the new point before it disappears behind L. In this case, add the point (x_i, yL) to the combined silhouette.

   (c) The new point's height AND its current group's height are below the height of the opposite group. For example, $yR = 1, yL =$ and the next point is $(x_i, 2) \in R$. In this case, do not add any points as the new point is completely covered up by the opposite group.

```python
# rectangles is a list of tuples (xL, xR, h)
def hidden_lines(rectangles):
    n = len(rectangles)
    if n == 0:
        return []
    elif n == 1:
        # return [(xL, h), (xR, 0)]
        return [(rectangles[0][0], rectangles[0][2]), (rectangles[0][1], 0)]
    else:
        # recursive steps
        L = hidden_lines(rectangles[:n//2])
        R = hidden_lines(rectangles[n//2:])

        # current height of Left and Right silhouettes respectively
        yL = -1*float('inf')
        yR = -1*float('inf')
        index_L = 0
        index_R = 0
        points = []
        while index_L < len(L) and index_R < len(R):

            # sort by x coordinate!
            if L[index_L][0] < R[index_R][0]:
                next_point = L[index_L]
                index_L += 1
                next_y = next_point[1]

                if next_y > yR:
                    points.append(next_point)
                elif yL > yR:
                    points.append((next_point[0], yR))

                yL = next_point[1]
            else:
                next_point = R[index_R]
                index_R += 1
                next_y = next_point[1]

                if next_y > yL:
                    points.append(next_point)
                elif yR > yL:
                    points.append((next_point[0], yL))

                yR = next_point[1]


        while index_R < len(R):
            points.append(R[index_R])
            index_R += 1

        while index_L < len(L):
            points.append(L[index_L])
            index_L += 1

        return points
```

Correctness

Finding the silhouette for a single rectangle is trivial. From there, as long as we can combine arbitrary silhouettes into one we will be able to reconstruct the full silhouette. In combining two silhouettes, L and R, note that the vertices of interest are created at the endpoints of the vertical edges of L and R. Without

loss of generality, consider a vertical edge of R and there are three possibilities when tracing the border of the silhouette. Either it the line begins above L and drops even or below to it, it begins below L and stretches above it, or it is completely contained in L. Each of these situations correspond to exactly one of the three recursive cases mentioned early, and thus any such vertical line can be represented. Thus, the combined silhouette can be generated.

Complexity

Define $C(n)$ to be the number of logical comparisons we perform during this algorithm. The base case simply returns, doing no further computations, and is thus $O(1)$. At each stage, there are two recursive calls, each passed approximately half of the list. This yields $2C\left(\frac{n}{2}\right)$ recursive work. Finally, at each level we can see there are around three to five logical comparisons for each element of either of the two subsets, yielding a workload of less than $5n$ comparisons at each level. Overall, this yields:

$$C(n) = \begin{cases} 1 & n == 1 \\ 2C\left(\frac{n}{2}\right) + 5n & n > 1 \end{cases}$$

Using the master theorem, we see that $a = 2, b = 2, c = 5, d = 1, k = 1$ and thus since $2 = 2^1 \rightarrow a = b^k$ we can see that this algorithm runs in $O(nlog(n))$ time.


```python
# rectangles is a list of tuples (xL, xR, h)
def hidden_lines(rectangles):
    n = len(rectangles)
    if n == 0:
        return []
    elif n == 1:
        # return [(xL, h), (xR, 0)]
        return [(rectangles[0][0], rectangles[0][2]), (rectangles[0][1], 0)]
    else:
        # recursive steps
        L = hidden_lines(rectangles[:n//2])
        R = hidden_lines(rectangles[n//2:])

        # current height of Left and Right silhouettes respectively
        yL = -1*float('inf')
        yR = -1*float('inf')
        index_L = 0
        index_R = 0
        points = []
        while index_L < len(L) and index_R < len(R):

            # sort by x coordinate!
            if L[index_L][0] < R[index_R][0]:
                next_point = L[index_L]
                index_L += 1
                next_y = next_point[1]

                if next_y > yR:
                    points.append(next_point)
                elif yL > yR:
                    points.append((next_point[0], yR))

                yL = next_point[1]
            else:
                next_point = R[index_R]
                index_R += 1
                next_y = next_point[1]

                if next_y > yL:
                    points.append(next_point)
                elif yR > yL:
                    points.append((next_point[0], yL))

                yR = next_point[1]


        while index_R < len(R):
            points.append(R[index_R])
            index_R += 1

        while index_L < len(L):
            points.append(L[index_L])
            index_L += 1

        return points
```

6 Extra Credit Problem 6

1. **Version 1**
   Since each of the submatrices have size $\frac{n}{2} \times \frac{n}{2}$, assuming that n is divisible by 2, then each multiplication above takes $\left(\frac{n}{2}\right)^3 = \frac{n^3}{8}$ multiplications. Since there are 8 such matrix products, overall this yields a total multiplication count of n^3, no improvement.

2. **Version 2**
   Define $M(n)$ to be the number of basic multiplications required to calculate the product of two $n \times n$ matrices using MMult recursively. Clearly, the base case of multiplying two 1×1 matrices requires just a single multiplication. We can see that every time we recursively call MMult, we turn a task of size n (multiplying two $n \times n$ matrices) into 8 tasks of size $\frac{n}{2}$ (multiplying two $\frac{n}{2} \times \frac{n}{2}$ matrices). Thus, the amount of recursive work must be $8M\left(\frac{n}{2}\right)$. At a given level, we perform no additional multiplications as we simply add the results of each subproblem, so we have no work at each level. Thus:

$$M(n) = \begin{cases} 1 & n == 1 \\ 8M\left(\frac{n}{2}\right) & n > 1 \end{cases}$$

3. **Solving with Master Recurrence**
   We can see this fits the master theorem with parameters $a = 8, b = 2, c = 0, d = 1, k = 0$ yields that:

$$8 > 2^0 \rightarrow a > b^k$$

   This implies that the algorithm is $O(n^{log_2(8)}) = O(n^3)$

4. **Solving with Algebra**
   Since the input size divides in two each time, we can see that it will take $log_2(n)$ levels of recursion to reach the base case. Thus:

$$M(n) = 8M(n/2) = 8^2M(n/4) = ... = 8^{log_2(n)}M(1) = 8^{log_2(n)} = 2^{3log_2(n)} = 2^{log_2(n^3)} = n^3$$

5. **Version 3**
   Since each of the submatrices have size $\frac{n}{2} \times \frac{n}{2}$, each matrix multiplication takes $\left(\frac{n}{2}\right)^3 = \frac{n^3}{8}$ scalar multiplications. Since there are 7 of these matrix multiplications, a total of $\frac{7n^3}{8}$ multiplications are required, a significant improvement on the obvious algorithm (although both are still $O(n^3)$).

6. **Version 4**
   Following the same logic as version 2, we can see that the number of scalar multiplications required using Version 4 recursively, $M(n)$, will be:

$$M(n) = \begin{cases} 1 & n == 1 \\ 7M\left(\frac{n}{2}\right) & n > 1 \end{cases}$$

   We can see this fits the master theorem with parameters $a = 7, b = 2, c = 0, d = 1, k = 0$ yields that:

$$7 > 2^0 \rightarrow a > b^k$$

   This implies that the algorithm is $O(n^{log_2(7)}) \approx O(n^{2.81})$, a significant improvement on the obvious algorithm.

7. In the base case, a multiplication of two 1×1 matrices, there are no additions. Recursively, the MMult splits the problem into 8 subproblems of size $\frac{n}{2}$ and at each recursive level performs 4 matrix additions with matrices of size $\frac{n}{2} \times \frac{n}{2}$. Each of these matrix additions requires $\left(\frac{n}{2}\right)^2 = \frac{n^2}{4}$ additions. In total then, there are n^2 scalar additions. Thus, if $A_{obv}(n)$ is the number of additions in the obvious algorithm:

$$A_{obv}(n) = \begin{cases} 0 & n == 1 \\ 8A_{obv}\left(\frac{n}{2}\right) + n^2 & n > 1 \end{cases}$$

   Applying the master theorem with $a = 8, b = 2, c = 1, d = 0, k = 2$ yields:

$$8 > 2^2 \rightarrow a > b^k$$

   which shows our obvious algorithm is again $O(n^3)$ when counting additions.

   Similarly for Strassen's method the number of additions is $15 * \frac{n^2}{4}$ so:

$$A_{Strass}(n) = \begin{cases} 0 & n == 1 \\ 7A_{Strass}\left(\frac{n}{2}\right) + \frac{15n^2}{4} & n > 1 \end{cases}$$

   Applying the master theorem with $a = 7, b = 2, c = 15/4, d = 0, k = 2$ yields:

$$8 > 2^2 \rightarrow a > b^k$$

   which shows our algorithm is again $O(n^{2.81})$ when counting additions.