- The assignment is due at Gradescope on Friday, Feb. 17 at 6:00p.
- You can either type your homework using LaTex or scan your handwritten work. We will provide a LaTex template for each homework. If you writing by hand, please fill in the solutions in this template, inserting additional sheets as necessary. This will facilitate the grading.
- You are permitted to study with up to 2 other students in the class (any section) and discuss the problems; however, *you must write up your own solutions, in your own words*. Do not submit anything you cannot explain. If you do collaborate with any of the other students on any problem, please do list all your collaborators in your submission for each problem.
- Similarly, please list any other source you have used for each problem, including other textbooks or websites. *Consulting problem solutions on the web is not allowed.*
- *Show your work.* Answers without justification will be given little credit.

PROBLEM 1 (35 POINTS) *Do K-T Chapter 5, Exercise 3 (on bank cards).*

Solution:
Input: A set of N cards where each card corresponds to a bank account.

Desired Output: Determine if there are more than $N/2$ cards in the set of N card which are all equivalent to each other, and if so return true.

```
Algorithm: ($S = \{1 \cdots N\}$)
    if $N = 1$ do:
        return true
    $S_o$ = Subroutine: ($S$)
    if $S_o = NULL$ do:
        return false
    return $|S_o| \geq \frac{|N|}{2} + 1$


Subroutine: ($S = \{1 \cdots N\}$)
    if $|S| = 2$ do: // first base case
        if first value in $S$ = second value in $S$ do: // equivalence testing
            return $S$ // all values in set are equivalent
        else:
            return NULL
    if $|S| = 1$ do: // second base case
        return $S$

    $H = \frac{N}{2} + 1$ // check for majority
// partition $|S|$ in half
    $S_1 = 0 \cdots \frac{N}{2}$
    $S_2 = \frac{N}{2} + 1 \cdots N$
    $S_1'$ = Subroutine: ($S_1$)

    If ($S_1' \neq$ NULL ) do:
        $c = |S_2|$
        $i = 0$
        // query through and test all values in $|S_2|$ against first value in $S_1'$
        while $i < c$ do:
            if the first value in $S_1'$ equals the $i$ value in $S_2$ do:
                $S_1' = S_1' \cup i$ value in $S_2$
            $i = i + 1$

        if $|S_1'| \geq H$ do:
            return $|S_1'|$
```

```
$S'_2$ = Subroutine: ($S_2$)
If ($S'_2 \neq$ **NULL** ) **do:**
    $c = |S_1|$
    $i = 0$
    // query through and test all values in $|S_1|$ against first value in $S'_2$
    while $i < c$
        if first value in $S'_2$ equals $i$ value in $S_1$ **do:**
            $S'_2 = S'_2 \cup i$ value in $S_1$
        $i = i + 1$

    if $|S'_2| \geq H$ **do:**
        return, $S'_2$

return **NULL**
```

Analysis:

Correctness:
We will prove by induction on k that **Algorithm: (**$S = \{1 \cdots N\}$**)** returns a correct output on any input $|S| = N$ with $N \leq k$.

For our base cases in our subroutine, we have $|S| = 1$ and $|S| = 2$. For $|S| = 1$, we always return S, because any set with one value will always have a more than $1/2$ cards equal to itself. For $|S| = 2$, we check that the cards are equivalent. If they are, we return both cards in the set, because we have a majority. If the two cards are not equal, we know that a majority is impossible, so we return null. Therefore, for $k \leq 2$, the algorithm always returns a correct output.

Induction: Assume that $k > 2$, and the statement is true for all values $k' < k$. Therefore, we will show it is correct for k. Consider some input $|S| = N$, where $N = k$ at most. If $N < k$, correctness is shown by induction. Therefore, assume $N = k$. Therefore, when we partition S into two halves, S_1 and S_2 directly after the base cases, we know that $|S_1|, |S_2| = \frac{k}{2}$, meaning that they are both correct solutions by inductive assumption. We now consider $N = k$.

Lemma 1**:** In order for there to be a set S, where $|S| = N$ with $\frac{N}{2} + 1$ cards that are equivalent, then either S_1 or S_2 made by partitioning S in half must have a majority equivalence set within the cards in their set, or $\frac{N}{4} + 1$ cards of equivalence in their set.

Proof by contradiction. Assume this is not true; then consider the best case, which would be the max amount of cards in an equal set which doesn't meet this requirement. Therefore neither side has a majority, but has exactly half of the set equal to the same value between the two sets. Then when we merge, we have $\frac{N}{4} + \frac{N}{4} = \frac{N}{2} < \frac{N}{2} + 1$. Therefore, at least one of S_1 or S_2 must have a majority equal set within their respective set in order to have a card for S to have a majority equal set in N.

Lemma 2: If Subroutine(S_1) and subroutine(S_2) are correct solutions and return if the sets have majority equal set for $S_1 = S \subset \{0 \cdots \frac{N}{2}\}$ and $S_2 = S \subset \{\frac{N}{2} + 1 \cdots N\}$, then Subroutine($S$) yields a correct solution for $S\{0 \cdots N\}$.

Proof: As we have that subroutine(S_1) and subroutine(S_2) are correct, and we return a set of majority equal cards if there is one, and **NULL** if there isn't one, and with insight from lemma 1 that we need a majority set on at least one side, we now analyze the cases of merging these two sets.

Both Null: If both values are null, then we know that no majority equal set exists in either set, and as proved above the combined set cannot be a majority equal set. Therefore we return **NULL** in the subroutine(S), and when we return to the algorithm it will return false.

Subroutine(S_1) $\neq$ Null and Subroutine(S_2) $=$ Null: If the first set has a majority value set, we query through all of S_2 comparing it to the first value in the equal majority set in S'_1. This happens at the line If ($S'_1 \neq$ **NULL**) **do:**. There cannot be more than 1 different values corresponding to a majority set made from S_1 in S'_1, by definition of a majority set, and we know that there is no majority set in S_2, then the only possible majority set formed by merging the two sets must be with the set of values in S'_1 equal to the values in S_2. Therefore we compare the first value in S'_1 to every value in S_2, adding each value to S'_1 that is equal as we progress through the list. Once we have finished the list, we compare the size of S'_1 to majority amount, in $|S'_1| \geq H$, to see if we have found a majority set. If this is true, we return S'_1, because it is a majority set for S. If $|S'_1| < H$, we know that no majority equal set can be formed in the set S, because the only potential majority set must from S'_1, and we tested it against every value in S_2 that could be added to create a majority set for S. Note that because S_2 has no majority set, and any other equivalence set from S_1 does not have a majority for S_1, because of the existence of the majority set in S'_1. Therefore in this case, we return **NULL**, which is correct in determining if S has a majority set.

Subroutine(S_1) $=$ Null and Subroutine(S_2) $\neq$ Null: This case follows the proof of the previous case, because it essentially has the same circumstance where we are merging two subsets, one with a equivalence majority set and the other without. All the logic is the same, and the same loop is performed but instead of comparing the first value in S'_1 to every value in S_2, which happens after the line "If ($S'_1 \neq$ **NULL**) **do:**" (it skips the first loop and comparison round), and instead it compares the first value in S'_2 to every value in

S_1 which happens after the line "If ($S_2' \neq$ **NULL**) **do:**". Because we switch which sets are being compared and which sets are null, the same process and logic follows, where any majority set in S must a majority set from S_2, which is S_2', which we return if "if $|S_2'| \geq H$ **do:**" and otherwise this can't be formed so then we know we have no majority set in S and return **NULL**.

Both not Null: In this case we will loop through both sides of the partition, comparing each produced set to the values not in that set and on the other side of the partition, or compare S_1' to all values in S_2 and S_2' to all values in S_1. If either of these sets produce a majority for $S\{0 \cdots N\}$ in the algorithm it will be returned to the entire algorithm, which we know is a majority set. If we don't return a majority set, because we know that S_1' and S_2' are the possible candidates for majority in S because they are majority equivalence sets in their subsets, we know no majority subset is possible so we return **Null**.

As every possibility is covered in merging the subsets and returns the correct answer for the respective merge, we know that the algorithm will always return true if there is a possible pair and false otherwise. Therefore, correctness is proved.

Efficiency:
The algorithm has some simple constants in the main body, and all the computing happens in the subroutine which it calls once. Therefore we will move to analyze the subroutine. In all of the base case work, there are only simple constant comparison operations, which are all $O(1)$. We move to forming the partition. Here, we form two new sets. This takes $O(N)$ time. Now we call the function recursively, but with half the size of the original index. Therefore we have $T(\frac{N}{2})$ for each time we call the subroutine. For the loop, we perform some constant operations and then iterate through every value in the opposite set of the one picked by the subroutine. Therefore this run time is $O(\frac{N}{2}))$. In the worst case scenario, both returned sets don't equal null, but the first set has not formed a majority equivalence set for S so we have called the subroutine twice and we enter both loops. Therefore, we have run time equals $2 \cdot T(\frac{N}{2}) + 2 \cdot O(\frac{N}{2}) + O(N)$ where the last $O(N)$ is the cost of forming the partition plus some $O(1)$ operations, which by the master theorem reduces to $O(n \cdot \log n)$ which is the desired run time and we are done.

PROBLEM 2 (35 POINTS) *(Inspired by The Family Circus.) Billy, a 7-year-old kid, is let loose in a park, which is an N-by-N grid of cells $(x, y) \in [N] \times [N]$. His starting location is the Southwest corner $(x, y) = (1, 1)$. He takes a walk in the park, always moving from a square to an adjacent square in one of the four cardinal directions. He never leaves the park, and never visits the same square twice. Eventually he ends up napping somewhere in the park. We are given the N-by-N matrix A, whose entries are $A[i, j] = \star$ if Billy never visits cell (i, j), $A[i, j] = \uparrow$ if Billy visited cell (i, j) and departed upwards (North direction); $A[i, j] = \leftarrow$ if Billy visited cell (i, j) and departed in the leftwards direction; similarly for $\downarrow, \rightarrow$; $A[i, j] = BILLY$ if Billy is napping in cell (i, j). As one example, $A[1, 1]$ will contain an arrow indicating the first step Billy took after entering the park (unless he stopped to nap immediately). Give an algorithm that makes only $O(N)$ queries to the matrix A, and locates Billy. Partial credit for $O(N \log N)$.*

Solution:
Input: An N-by-N matrix, A, which represents the coordinates of a park, which entries are either un-visited, or have an arrow pointing to the next step taken by Billy, or contain Billy.
Desired Output: Billy's coordinates (i, j) in $O(N)$ queries.

General Approach: Create a set of edges of a "box" which indicates where Billy could be. With each recursive step. draw a line half way across the box and remove whichever half doesn't contain Billy, found by analyzing arrow direction. This reduction is achieved by setting the edge which contains the space that doesn't have billy to be equal to the half way line drawn. Alternate between horizontal and vertical lines with each recursive call to reduce overall amount of iteration steps needed. Return billy when encountered.

Subroutine and Algorithm:

Algorithm (*N-by-N matrix*):

```
if $N = 1$ do:
    return $(1, 1)$
// create sides of box
$E_L, E_B = 0$
$E_R, E_T = N$
return Subroutine($\frac{N}{2}$,$E_L$, $E_R$, $X$):
```

```
Subroutine ( $C$, $P$, $Q$, $D$):
    $Ar_c = 0$  //arrow counter
    If $D = X$ do:
        $i = P, j = C$
        while $i \leq Q$ do: // draw line loop
            if $A[i,j] = BILLY$ do:
                return $(i,j)$
            if $A[i,j] = \downarrow$ do:
                $Ar_c = Ar_c - 1$
            if $A[i,j] = \uparrow$ do:
                $Ar_c = Ar_c + 1$
            $i = i + 1$

        if $Ar_c > 0$ // BILLY ABOVE
            $E_B = C$ // bottom edge becomes former mid line
        if $Ar_c \leq 0$ // BILLY BELOW
            $E_T = C$ // top edge becomes former mid line

        Subroutine($\frac{Q-P}{2}$,$E_B$, $E_T$, $Y$):

    If $D = Y$ do:
        $i = C, j = P$
        while $j \leq Q$ do:
            if $A[i,j] = BILLY$ do:
                return $(i,j)$
            if $A[i,j] = \leftarrow$ do:
                $Ar_c = Ar_c - 1$
            if $A[i,j] = \rightarrow$ do:
                $Ar_c = Ar_c + 1$
            $j = j + 1$

        if $Ar_c > 0$ // BILLY RIGHT
            $E_L = C$ // left edge becomes former mid line

        if $Ar_c \leq 0$ // BILLY LEFT
            $E_R = C$ // right edge becomes former mid line

        Subroutine($\frac{Q-P}{2}$,$E_L$, $E_R$, $X$):
```

Correctness:

Feasibility: We prove by induction on K that **Algorithm (*N-by-N matrix*)**: outputs the location of Billy for any input N with $N \leq K$.
Base case: As $N = 1$ returns $(1,1)$, and this is the only possible place for Billy, because we have only one space, we know the algorithm return the correct answer in $O(N)$ time for $N = 1$.

Inductive step: assume $1 < K$ and statement has been proved for all values $K' < K$. We'll show it for K. Consider N of size at most K. If $N < K$ then correctness for N has already been shown, so assume now that $N = K$.

In the execution of the algorithm, first we check our base case, and then set up a "box" representing the dimensions of the matrix. Our box has respective edges $E_L, E_B = 0$ and $E_R, E_T = N$, where we have edges representing left, right, bottom, and top of a rectangle which will always maintain Billy, and where their value corresponds to the coordinate they lay on. The scheme is as follows: E_L starts at 0, so it's $X = 0$, whereas E_R starts at N, so it's $X = N$. E_B starts at 0, so it's $Y = 0$, whereas E_T starts at N, so it's $Y = N$.

I claim that the box will always represent the dimensions of possible spaces where billy could be, and we reduce the area at every iteration by half. We do this through drawing a line in the middle and determining which side that Billy is on. Therefore, we start our first subroutine iteration with $C = \frac{N}{2}$, $P = E_L$, $Q = E_R$, and $D = X$. These variables are defined as follows: P, Q indicate the starting and ending points of the straight line we draw. C is the constant value we keep for each query in the line. D is either X or Y and will determine if we draw a line horizontally or vertically. Note that this will influence if C is a constant for x or y. In other words, for the first call we draw a horizontal line as $D = X$, starting at the left edge and ending at the right edge, with the constant being the middle point of N. We always choose the middle point, as this partition gives us the most possible space to eliminate with each iteration, knowing that Billy could be on either side of the partition.

We then enter the subroutine, which will call one loop iteration of length $Q - P$, change the length of an edge (either E_B or E_T) and then call the subroutine again. An important feature is that the length of an edge is changed to the current constant C. As every iteration calls C as the midpoint of the line $Q - P$, and then sets the coordinates of the line to C, we know that it reduces the area of the box by one half of its previous size per iteration. This is shown as follows.

Take the first subroutine, which has $C = \frac{N}{2}$. It will either set E_B or E_T to this value, based on which side it determines Billy to be. Because we define the matrix as the area between the box sizes, setting either of these to $\frac{N}{2}$ will reduce the area by a halve. As we start with a matrix with size $N \cdot N$, and the length is determined be the differences between the two dimensions of the box, we get $N \cdot \frac{N}{2}$ after every iterations, and we know $N \cdot \frac{N}{2} < K \cdot K$, so by inductive assumption we know that the algorithm finds Billy in a matrix of this dimensions as it is smaller than K and assumed.

However, much of this proof hinges on **1)** knowing where Billy is initially and **2** not losing Billy in any iteration, because we don't want to remove a group of coordinates where he could be. First, we are given where he is initially by the problem, so we only need to worry about not losing him, or in other words show that we don't remove an area that he could be in.

LEMMA 1: As Subroutine($\frac{Q-P}{2}$,E_B, E_T, Y) correctly returns Billy's location then Subroutine($\frac{N}{2}$,E_L, E_R, X) also correctly returns Billy's location.
The difference between these two calls, is that we have reduced the size of the box by setting $E_B = C$ or $E_T = C$, which consequently reduces the distance between P and Q for the next call. Therefore, we will show that this is correct. First, if we encounter Billy on the line drawn in the subroutine, we return Billy's coordinates. Then correctness is complete. If we don't find Billy, we choose to change the value of E_B or E_T. Consider the following cases where we don't find Billy:

No Arrow is encountered on the line: Then we reduce E_T to C. If no arrow is encounter and Billy starts in the South-west corner as given, and we cross the entire side of the matrix, we know that Billy must be below the line, because otherwise there must have been some point Billy crossed the line if Billy was above it. Therefore, it is impossible for Billy to be in any area of the matrix above the line and we can reduce the top of the matrix to be that line, and then we call the recursive call which we know finds him by inductive hypothesis. Therefore, this case correctly finds Billy.
An equal amount of arrows are encountered on the line: Then Billy has crossed from below to above the line back and forth an equal amount of times, meaning he must be at the same side that he started at, because there was no point where he crossed the line and did not cross back. Therefore, we know he cannot be in the upper half of the matrix, so we can reduce the top of the matrix to be the half way line, and then we call the recursive call which finds him by inductive hypothesis. Therefore, this case correctly finds Billy.
More up arrows are encountered then down arrows on the line: Then Billy has crossed from below to above the line back and forth an equal amount of times, with the addition of at least one more crossing to the upper half. Therefore, he cannot be in the lower half, because each time he returned he went back to the upper half at least once. Therefore, we can reduce the bottom half of the matrix to be that line, and then we call the recursive call which finds him by inductive hypothesis. Therefore, this case correctly finds Billy.
More down arrows are encountered then up arrows: This case cannot happen because knowing that Billy starts below the line, to produce a single down arrow along the line, Billy must be above the line, and in order to get above the line he must cross from the bottom at least once, meaning for every down arrow there is at least one up arrow. Therefore, there can never be more down arrows than up arrows, and we don't need to consider this case.

Therefore we have shown that considering every amount of arrows, we can correctly reduce either the bottom or top edge, based on where Billy isn't and then call the subroutine which will find him by inductive assumption.

Note that we apply the same process of counting arrows for right and left cases when considering the execution of $D = Y$, which has the same properties and logic checking to see if Billy is above and below but applied for vertical lines instead of horizontal lines. We maintain the same knowledge that Billy starts in the bottom left corner and follow him accordingly, only removing areas that we know can't contain him. I am not providing this proof because it is repetitious and all the components are identical by shifted to the vertical axis.

Therefore, in all cases, the algorithm will either return the position of Billy or a smaller matrix which we know contains Billy, which we can then find him from inductive hypothesis. This proves correctness.

Efficiency: Begin with Algorithm, and assume $N \neq 1$ (in which case we end with $O(1)$ or $O(N)$ run time and are done). Therefore, we enter X subroutine. This subroutine performs some $O(1)$ operations in setting up constants initially, and then executes a while loop through a straight line of i variables, which is $O(n)$ for the first iteration. Inside the while loop, we check for Billy (if we find Billy we terminate, which will be worst case $O(N)$ if found on the first line), then we check for up and down arrows (we record this in arrow counter), and then increment through the loop until we have reached Q. These are all $O(1)$ run time operations, and as $P - Q$ starts at length N the while loop takes $O(N)$ in the first (and it only decreases length from there as we reduce a box edge side each iteration). Note also that because we alternate between drawing lines horizontally and vertically, we maintain $O(N)$ for each iteration. If we were to only draw a line horizontally from the midpoint and not alternate, we would half the size of our matrix each time but not halve the amount of queries proportional to that matrix. Hence the alternating, which is the same amount of reduction for less queries. Because either the bottom of top of the box is set to the midpoint of the previous distance between the two, we have $O(\frac{N}{2})$ possible queries for the next iteration and reduced the overall potential size of the matrix by a half. Therefore, for the next iteration we have $T(\frac{N}{2}) + O(N)$ run time. Then we initiate the subroutine again, drawing a vertical line. This time the subroutine has *nearly* the exact same behavior as previous iteration, other than it queries through a straight line of j variables instead of i variables. However, at its most, it will perform $O(N/2)$ iterations in the while loop, as the edge distance is already halved. From there, we halved the area again and continue iterating. Therefore, for any respective of the subroutine, we have worst case behavior of $T(\frac{N}{2}) + O(N)$ run time. Note that the $O(N)$ aspect of drawing a line to partition the middle of matrix functions as the cost of "combining" of two solutions, because it removes half of the previous matrix and ensures Billy is in the half we keep, promising we can always find him. This is possible because we always know the origin point of Billy, and through tracking his path we can always determine which sector he is in, therefore narrowing down his location and combining the work from each recurrence. From the master theorem, we have $T(\frac{N}{2}) + O(N) = O(N)$. Therefore, we have run time $O(N)$.