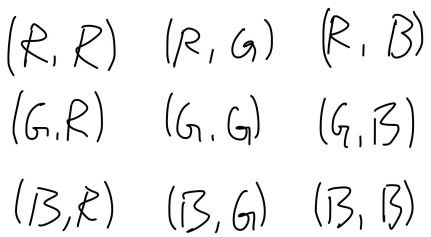
CSC265 Fall 2020 Homework Assignment 8

Student 1: Andrew Feng
Student 2: Kunal Chawla

1. Consider the following simplified version of the Mastermind game.
   The Chooser chooses a sequence of two (not necessarily different) colours from among {Blue, Green, Red}. The Guesser must discover this sequence.
   Each turn, the Guesser gives the Chooser a sequence of two (not necessarily different) colours from among {Blue, Green, Red}. Then the Chooser responds with the number of positions in which the two sequences agree.
   The game ends when the Chooser answers 2, indicating that the Guesser has discovered the sequence.

   (a) Prove that any decision tree for this problem has height at least 4.
   If you can't solve this problem, you will get a small number of marks for proving that any decision tree for this problem has height at least 3.
   Solution:
   The chooser stores a 3 by 3 array of $\{R, G, B\}$ pairs, where the first row contains $(R, R), (R, G), (R, B)$, the second $(G, R), (G, G), (G, B)$, and the third $(B, R), (B, G), (B, B)$:



   This way any query the guesser correspond to an entry of the array.
   For any color $C \in \{R, G, B\}$, the row where pairs have C in the first slot is called row C, and the column where pairs have C in the second slot is called column C. Let $(i_k, j_k) \in \{R, G, B\}^2$ be the kth query made by the guesser.
   Initially, there are 9 possible final sequences. When there is only one possible final sequence and the guesser guesses that sequence, the chooser must answer 2 to be consistent. When there are more than one possible final sequence, the chooser can answer < 2 and still be consistent. We will show the earliest the chooser must answer 2 is after the third query.
   For the first query, the chooser answers 1. This is consistent since there are 9 possibilities. The answer 1 means the final sequence must be in the row i_1 or in column j_1, but must not be (i_1, j_1) itself, leaving 4 possibilities.
   Now for the second query. Suppose first that ($i_2 \neq i_1$ and $j_2 \neq j_1$) or ($i_2 = i_1$ and $j_2 = j_1$). In this case the chooser answers 0, and this is consistent because the final sequence must be in row i_1 or column j_1. The chooser ends up with 4 possibilities because it does not eliminate any possibilities. Now suppose that (i_2, j_2) shares row or

column but not both with (i_1, j_1), and the chooser answers 0. This is consistent because among the 4 possible sequences, the two on row i_1 share no entry with those on column j_1. The chooser ends up with 2 possibilities since the answer 0 would imply the final sequence is (on row i_1 but is not (i_1, j_1)) or (on column j_1 but is not (i_1, j_1)).

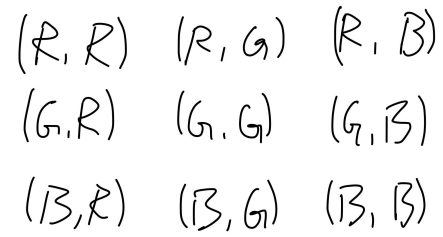
On the third query, the chooser has at least 2 possibilities by last paragraph. So the chooser does not have to answer 2 to be consistent (if (i_3, j_3) is one of the possibilities, the chooser answers 1; otherwise the chooser answers consistently). This means the earliest the chooser must answer 2 is after the third query. So in order for the chooser to answer 2, there must be at least 4 queries, giving a decision tree of height at least 4.

(b) Give a decision tree for this problem that has height 4. Justify why your algorithm is correct.

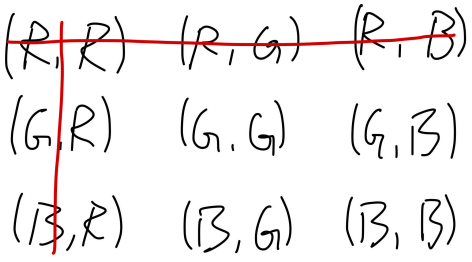
Solution:

We will continue using the array previous setup. And the algorithm is as follows

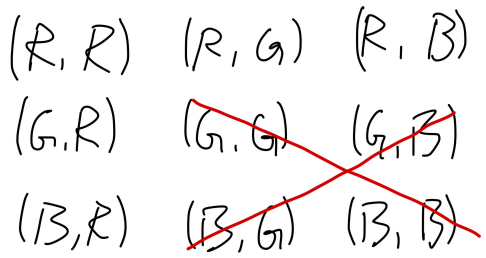
1) First we compare x and (R, R): If the chooser answers 0, this means the final sequence does not contain R but instead consists of G, B, or both, and we execute step 2. If the chooser answers 1, this means the final sequence contains one R, and we execute step 3. If the chooser answers 2, we are done—this took 1 query.



2) We compare x and (G, G): If the chooser answers 0, this means the final sequence does not contain R and G, so we know the sequence is (B, B). We compare x and (B, B), and the chooser must answer 2—this took 3 queries in total. If the chooser answers 1, we know the final sequence contains G but is not (G, G), we execute step 4. If the chooser answers 2, we are done—this took 2 queries in total.



3) We compare x and (G, R): If the chooser answers 0, this means the final sequence contains R but only in the first slot, and we execute step 5. If the chooser answers 1, we know the final sequence contains R in the second slot but is not (G, R). So the final sequence must be (B, R). We compare x and (B, R) and the chooser must answer 2, and we are done—this took 3 queries. If the chooser answers 2, we are also done—this took 2 queries.



4) We compare x and (B, G): If the chooser answers 0, we know the final sequence contains a G, does not contain R, is not (B, G). So it must be (G, B). We simply compare x and (G, B), and the chooser must answer 2, so we are done—this took 4 queries. The chooser cannot consistently answer 1 because answer 1 to both (G, G) and (B, G) implies the sequence is (R, G), which contradicts the answer 0 to (R, R) in the first place. If the chooser answers 2, we are done—this took 3 queries.

5) We compare x and (R, G): The chooser cannot answer 0 consistently because we know the final sequence contains R in the first slot. If the chooser answers 1, we know the final sequence contains R only in the first slot, does not contain G, so it must be (R, B). We compare x and (R, B), and the chooser must answer 2, so we are done—this took 4 queries. If the chooser answers 2, we are also done—this took 3 queries.

In any case above, at most 4 queries are used.

2. Consider the problem of searching for a value x in an $n \times n$ matrix A of integers in which the entries of each row are sorted in nondecreasing order from left to right and the entries of each column are sorted in nondecreasing order from top to bottom.
Prove that every algorithm solving this problem which only performs comparisons between x and elements of A must perform at least $2n - 1$ comparisons in the worst case.

Solution: We use an adversary strategy to prove the lower bound in question. First we define some things we use in the proof

For any sequence $A[i_1, j_1]$, $A[i_2, j_2]$, ... of comparisons made by the algorithm, and for some kth step in this sequence, the adversary maintains a set R_k of indices which are 'ruled out'. We define R_k recursively as follows:

(a) Initially, $R_0 = \varnothing$

(b) After the $k + 1$st comparison is made to some $A[i_{k+1}, j_{k+1}]$, we let R' be the set of elements which are now ruled out:

i. If the comparison returns $x = A[i_{k+1}, j_{k+1}]$, then R' is the set of all elements $A[i', j']$ in the array where either $i' \neq i_{k+1}$ or $j' \neq j_{k+1}$.

ii. If the comparison returns $x > A[i_{k+1}, j_{k+1}]$, then R' is the set of all elements $A[i', j']$ in the array where both $i' \leq i_{k+1}$ and $j' \leq j_{k+1}$

iii. If the comparison returns $x < A[i_{k+1}, j_{k+1}]$, then R' is the set of all elements $A[i', j']$ in the array where both $i' \geq i_{k+1}$ and $j' \geq j_{k+1}$

Then $R_{k+1} = R_k \cup R'$.

We claim that given some correct algorithm that halts after k steps, then after the kth step either a comparison returns $x = A[i_k, j_k]$ or R_k is equal to the whole array.

Indeed, suppose that the algorithm halts after k steps and none of the comparisons returned $x = A[i_k, j_k]$, then this means that x was not found in the array. Therefore we know that every element in the array must have been ruled out at some point. In other words, for each pair of indices i, j the algorithm must have made a comparison to some $A[i', j']$ such that either

- The comparison returned $x < A[i', j']$ and $i \geq i'$ and $j \geq j'$,
- The comparison returned $x > A[i', j']$ and $i \leq i'$ and $j \leq j'$.

Indeed, suppose for contradiction that the algorithm returns while not all positions in the array have been ruled out. Then there exists some i, j such that $A[i, j]$ does not satisfy the two conditions we just outlined. Therefore since the algorithm returns that x is not in the array, then this may be incorrect because it is possible that $A[i, j] = x$.

It follows by the definition of R_k that R_k is equal to the whole array.

Therefore we concoct an adversary strategy that never responds '=' to any comparison and such that R_k is not the whole array for any $k < 2n - 2$. This would imply that any correct algorithm makes at least $2n - 1$ comparisons.

Consider the following adversary strategy:

- If the algorithm makes a comparison to $A[i, j]$ where $i = j$, then the adversary responds $>$.
- If the algorithm makes a comparison to $A[i, j]$ where $i < j$, then the adversary responds $>$.
- If the algorithm makes a comparison to $A[i, j]$ where $i > j$, then the adversary responds $<$.

The idea is that the adversary is trying to delay when the elements near the diagonal (from bottom left to top right) get ruled out. For example, in the case $n = 3$ the following table shows how our adversary responds to each comparison of the form x '?' $A[i, j]$:

$$\begin{bmatrix} > & > & > \\ > & > & < \\ > & < & < \end{bmatrix}.$$

Now, observe that any element on the diagonal $A[i, i]$ gets ruled out if and only if the algorithm makes a comparison to $A[i, i]$. Also, observe that any element immediately below the diagonal gets ruled out if and only if that element gets compared.

Finally, observe that as there are n elements on the diagonal and $n - 1$ immediately below the diagonal, then this forces the algorithm to make $2n - 1$ comparisons.

Indeed, suppose for contradiction that there exists some algorithm which makes at most $2n - 2$ comparisons. Then as our adversary never returns '=' to any query, by the reasoning above we know that all elements of the array are contained in R_{2n-2}, however by the previous

paragraph, as there were at most $2n-2$ comparisons, then one element on the diagonal or off-diagonal is not in R_{2n-2}, a contradiction.

Therefore any algorithm solving the problem must perform at least $2n-1$ comparisons in the worst case.