CSC373 – Problem Set 2

Remember to write your **full name(s)** and **student number(s)** prominently on your submission. To avoid suspicions of plagiarism: at the beginning of your submission, **clearly state any resources (people, print, electronic) outside of your group, the course notes, and the course staff, that you consulted**.

Remember that you are required to submit your problem sets as both LaTeX `.tex` source files and `.pdf` files. There is a 10% penalty on the assignment for failing to submit both the `.tex` and `.pdf`.

Due Oct 3, 2020, 22:00; required files: ps2.pdf, ps2.tex

Answer each question completely, always justifying your claims and reasoning. Your solution will be graded not only on correctness, but also on clarity. Answers that are technically correct that are hard to understand will not receive full marks. Mark values for each question are contained in the [square brackets].

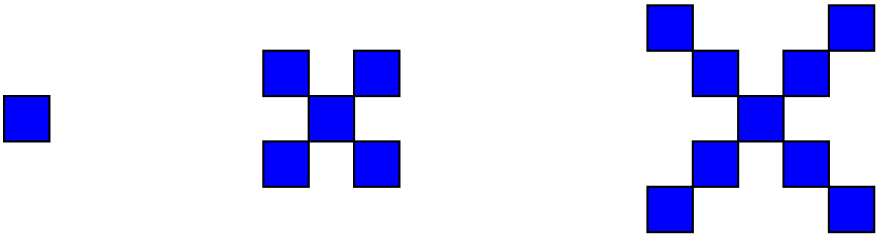
You may work in groups of up to THREE to complete these questions.

Please see the course information sheet for the late submission policy.

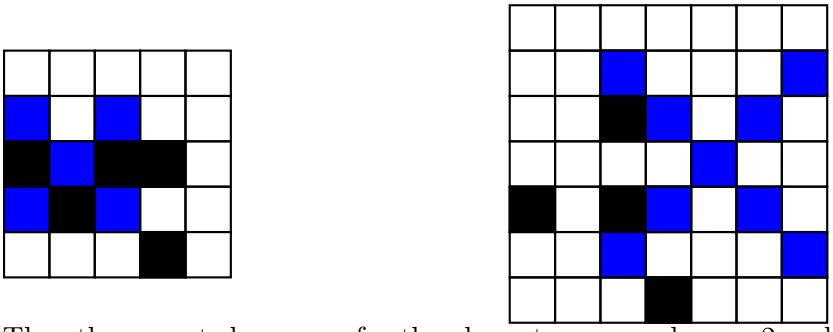
[15 points]

Sushant has played a lot of Tic-Tac-Toe in his early years, and his favorite mark was $\times$. So whenever he sees a grid, he's looking to find the biggest $\times$ symbol he can fit in it.

You are given an $n \times n$ array with entries in 0 or 1. 0 indicates that that location is empty, and 1 indicates that the location is filled. Your goal is to determine the largest size of an $\times$ symbol that can be filled in the grid while utilizing only the 0 locations. The following figures below demonstrate $\times$ with sizes 1, 2, and 3 respectively (a zero sized $\times$ would have no blue squares).



In the example grids drawn below, the squares with 1 in the input array are filled with black color, and all the squares colored blue or white are 0 locations. The blue squares visualize an $\times$ with the biggest possible size that can be fit inside the grid using only non-black locations.



Thus the expected answers for the above two examples are 2 and 3.

Design an algorithm for the above problem with a worst case running-time complexity of $O(n^2)$ following the steps below.

1. **[1 point]** Clearly and precisely specify in English the problem you wish to solve.

   The main problem is **"What is the largest size of an X that can be filled in the grid while utilizing only the 0 location."**

2. **[4 points]** Give a recursive solution for the problem (including base cases) and justify it. (Hint: You may need to define more than one recursive function)

   Helper functions:

   The helper functions take 2 inputs, which represents the coordinates(index) of the position we are looking at.

   Bottom Right(BR):

   This helper function tries to find the length of the longest continuous diagonal line from the current position to the top-left corner.

   Base case:

   Our base case for the main function contains 2 possible scenarios:

   (1) When there is 1 at (a, b); which means the longest length of the continues diagonal lines from the current position to upper-left corner will be 0 since 1 means this tile is occupied.

   (2) When there is 0 at (k,0) or (0, k) (where $0 \leqslant k \leqslant n-1$); this means this tile is close to the boundary of the board, and the length of the longest continuous diagonal lines from the current position to upper-left corner will have length of 1 since their upper-left position is out of bound.

   Recursive case:

   We can find the length of the **longest continuous diagonal line(LCDL)** of the current tile by finding the length of the LCDL of the upper-left tile, and adding 1 to it.

$$\text{BR(a , b)} = \begin{cases} 0 & \text{if value at (a,b) is 1, or a} < 0\text{, or b} < 0 \\ 1 + P(a-1, b-1) & \text{if a} \geqslant 0 \text{ and b} \geqslant 0 \end{cases}$$

   Bottom Left(BL):

   This helper function tries to find the longest length of the continues diagonal lines from the current position to top-right corner.

   Base case:

   Our base case for the main function contain 2 possible scenarios:

   (1) When there is 1 at (a, b); which means the longest length of the continues diagonal lines from the current position to upper-left corner will be 0 since 1 means this tile is occupied.

   (2) When there is 0 at (k,n) or (0, k) (where $0 \leqslant k \leqslant n-1$); this means this tile is close to the boundary of the board, and the length of the longest continuous diagonal lines from the current position to upper-left corner will have length of 1 since their upper-left position is out of bound.

   Recursive case:

   We can find the length of the **longest continuous diagonal line(LCDL)** of the

current tile by finding the length of the LCDL of the upper-right tile, and adding 1 to it.

$$\text{BL(a , b)} = \begin{cases} 0 & \text{if value at (a,b) is 1, or a} < 0\text{, or b} > \text{n} \\ 1 + P(a-1, b+1) & \text{if a} \geqslant 0 \text{ and b} \leqslant \text{n} \end{cases}$$

The main function:

Base case:
Our base case for the main function contains 2 possible scenarios:
(1) When array is empty or there is no 0 in the array: This means that there is no position for us to filled in the X which means the largest size of the X has to be 0.
(2) When n<3, in this case since (1) already passed the no zero cases, they only size can contain in a array smaller than $3 * 3$ will be size 1

For loop:
The for loop will go over the entire array. At each point (a , b) we will look for the longest length of the continues diagonal lines from the current position to upper-left corner first.

If we find a point which has the length larger than or equal to 3 (We found the right bottom point of X). We will check the point at index (a, (b - length to upper-left + 1) (which is the bottom-left point of the X (if there exist a X).

We are looking for the length of the longest continuous diagonal line from the current position to upper-right corner. If they have the same length, then it means there exist a X with size length // 2 + 1; else, we can still make a "X" with size min(bottom-left point's length, bottom-right point's length)//2 + 1.

After going through every position we keep the biggest size of X. And this is the value we are looking for.

Let BR-len = "length of the longest continuous diagonal line to top-left corner from current tile",
and BL-len = "length of the longest continuous diagonal line ot top-right corner from current tile".

$$\text{T(a , b)} = \begin{cases} 0 & \text{if the assumed bottom-left point of "X" is out of bound} \\ min(BR-len, BL-len)//2+1 & otherwise \end{cases}$$

3. **[1 point]** Specify all the subproblems that your algorithm needs to solve.

The sub-problem is **"What is the longest length of the continues diagonal lines from the current position to (upper-right/upper left) corner"**.

4. **[3 point]** Specify the memoization data structure(s), clearly define what value will be stored in each location at the end of the algorithm, and give a good bottom-up evaluation order for filling the memoization datastructure(s).

We will be using **dictionary (hashtable)** as our memoization data structure.

The point to note about dictionary(hashtable) is that it has constant look-up time while having a linear space complexity.

By storing the length of the longest diagonal line as value, and the tile's coordinate as key, we avoid the need to look for previous(upper-left/upper-right)tile's length which may cause repetitive calls issue.

With 2 dictionaries(hashtables), we can simply store the value, and when we wish to find the length of the longest diagonal line of the current tile, we only need to look up the length of the tile that is to the upper-left/upper-right of the current tile(which is stored in our dictoinary already) and add 1 to its length.

Also, by building up the 2 dictoinaries from the top-left corner, instead of starting at the bottom-right corner, we are solving the problem with the "bottom-up" evaluation order, because we will be solving our bases cases first.

5. **[6 point]** Write down the final dynamic-programming algorithm (non-recursive), and analyze its time and space complexity.

```
def largest(a):
    n <= length of a
    maximum <= -1
    bottom_right_dict <= {}
    bottom_left_dict <= {}

    # build bottom_right_dict
    for every row i in a:
        for every tile j in row i:
            if a[i][j] is 1:
                bottom_right_dict[(i,j)] = 0
            elif the upper-left tile(a[i-1][j-1]) is out of bound:
                bottom_right_dict[(i,j)] = 1
            else:
                bottom_right_dict[(i,j)] = 1 + bottom_right_dict[(i-1,j-1)]

    # build bottom_left_dict
    for every row i in a:
        for every tile j in row i:
            if a[i][j] is 1:
                bottom_left_dict[(i,j)] = 0
            elif the upper-right tile(a[i-1][j+1]) is out of bound:
                bottom_left_dict[(i,j)] = 1
            else:
                bottom_left_dict[(i,j)] = 1 + bottom_left_dict[(i-1,j+1)]

    # find total number of "X"
    for every row i in a:
```

```
        for every tile j in a:
            BRD_val <= buttom_right_dict[(i,j)]
            if BRD_val is not 0,and (i-(BRD_val -1),j) is not out of bound:

                BLD_val <= buttom_left_dict[(i-(BRD_val -1),j)]
                contender <= min(BRD_val, BLD_val) // 2 + 1

                if contender > maximum:
                    maximum <= contender

    return maximum
```

Time complexity:

Suppose we have a n*n grid A. The build bottom right dict loop will iterate through the array exactly once which takes $O(n^2)$ time, and for each iteration it takes constant time to consider the length the block itself(check does this block consist a 0) and add the previous diagonal block's value. So the total cost time is $O(n^2)$.

The build bottom left dict loop has the same condition with above loop, so the time complexity is also $O(n^2)$

For the find largest size of "X" loop, it goes over A exactly once, and the if statements check only takes constant time per iteration. So this loop will take at most $O(n^2)$ time.

The three loops have the same time complexity of $O(n^2)$, so the total time complexity is also $O(n^2)$

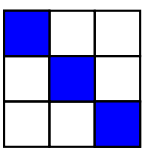
Space complexity:

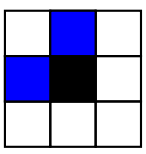
The space complexity of our code is O(n^2).

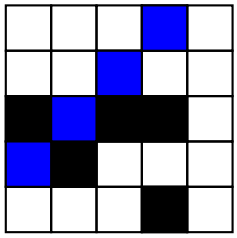
We have 2 dictionaries, each with the same size as the given $n * n$ array. And it's the only memorization structure we have in the function.

Therefore it will take O(n^2) space.

Hint: As a warmup, you can try to first solve the problem for finding the largest possible diagonal line, either $\diagup$ or $\diagdown$ can be fit in the 0 locations. Consider the following example grids:







Thus the expected answer for these three examples are 3, 2, and 4 respectively.