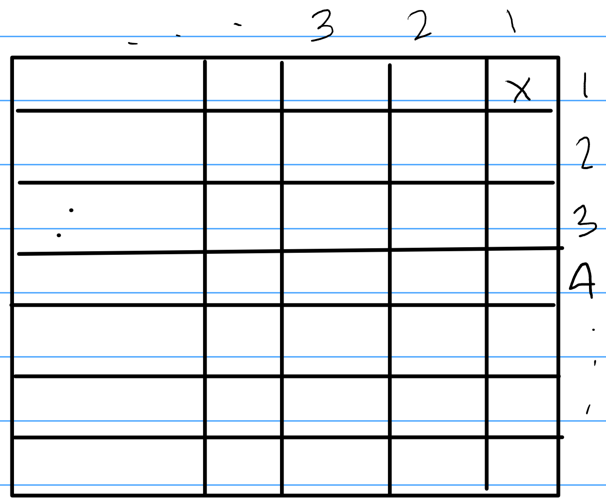
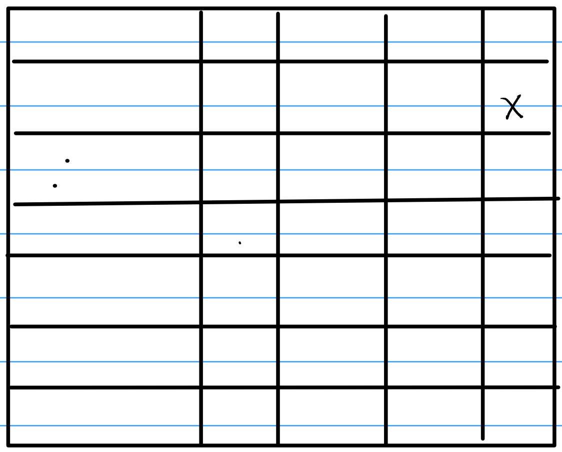
CSc 335 Last Lecture
(Quiz on Tuesday)

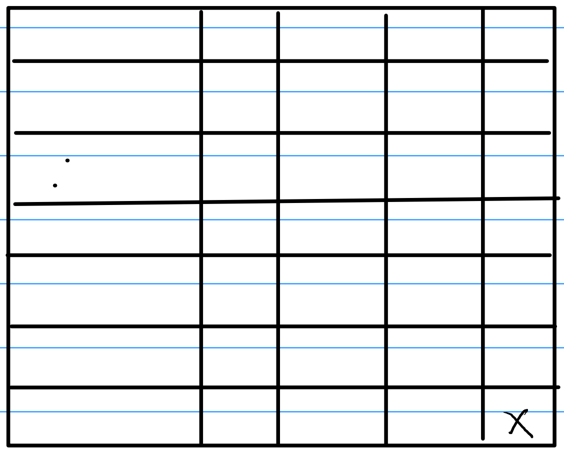
1. Discussion of Queens Problem (HW)
2. Discussion of Matrix Problems (HW)
3. Recursion in TLS Scheme

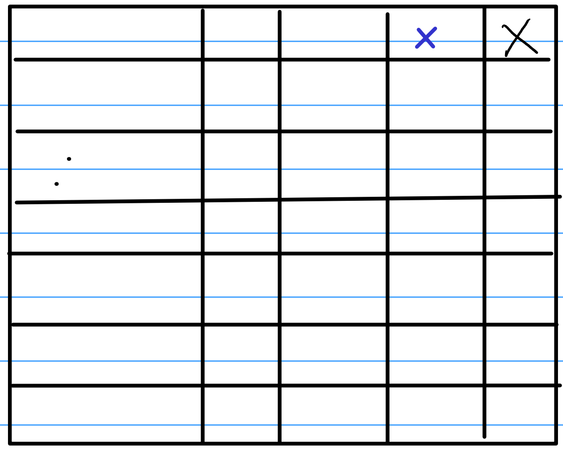
1.



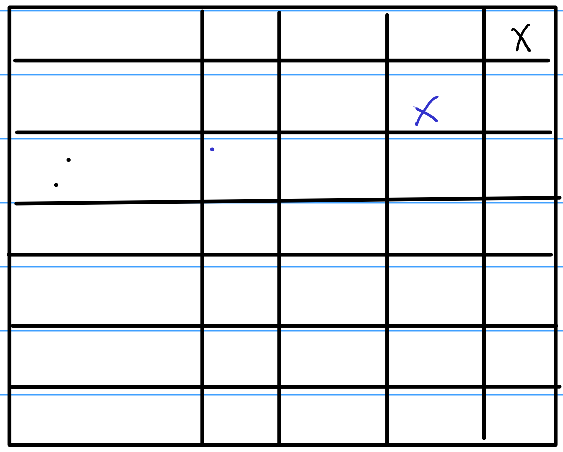
I call the various placements of a single queen in the rightmost column a 1-configuration



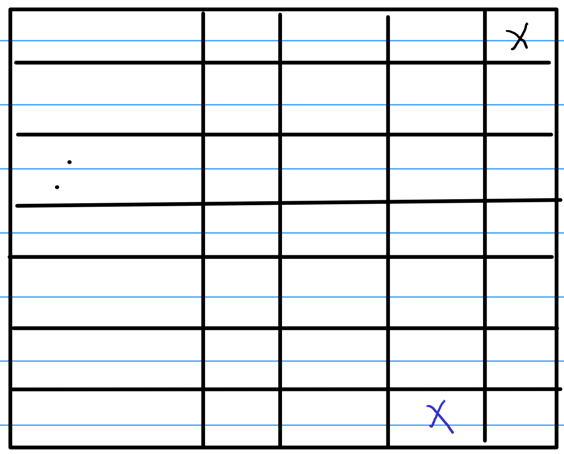




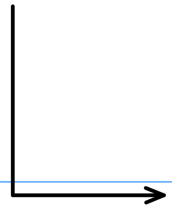
→ unsafe, so filtered out

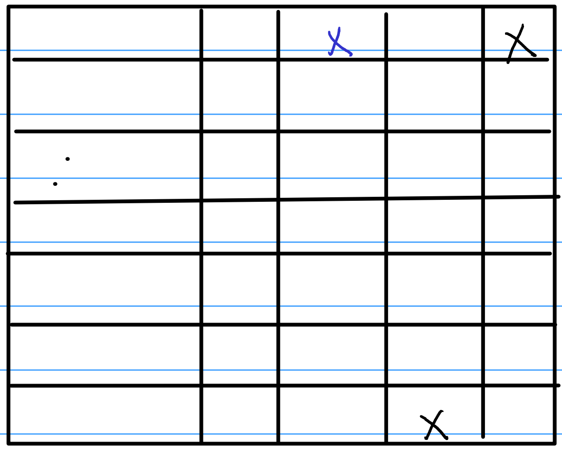


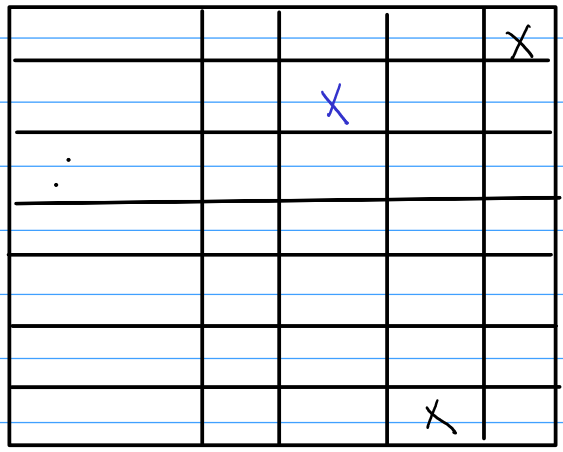
→ unsafe, so filtered out

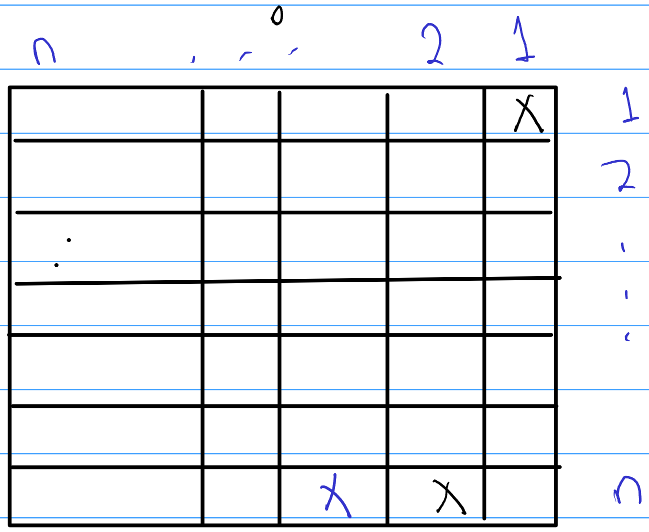


→ safe, so retained — we keep all so far safe 2-configurations for further processing







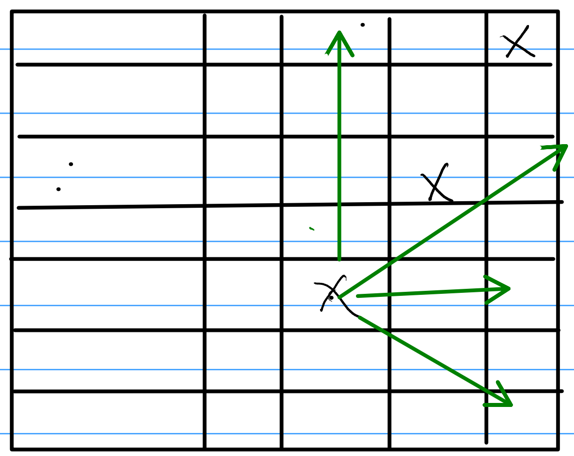


Again, generate all 3-configurations with queens held fixed in positions (0 0) and (0 n-1)

Then filter for safety – keeping only those 3-configs in which no queen is attacking another.

Etc. Do this for ALL 8 possible 1-configs ...

What's involved in checking the safety of a configuration?



This is a safe 3-config since there are no queens vertically, horizontally or diagonally from any of the already placed queens.

Even If you don't have time to work out your safe? predicate, you will very much want to study the A&S code to understand the need use of flatmap. Especially observe how the IH/IS are simplified by its use.

On now to the matrix problems — at least some of them

```
; matrix operations
; first, some data -- matrices as sequences of rows
(define m '((1 2 3 4) (4 5 6 6) (6 7 8 9)))
(define n '((1 2) (3 3) (4 5) (6 7)))
(define v '(1 2 3 4))
(define w '(5 6 6 7))
```

$$- \begin{bmatrix} 1 & 2 & 3 & 4 \\ 4 & 5 & 6 & 6 \\ 6 & 7 & 8 & 9 \end{bmatrix}$$

```
; here is a function given by the authors
(define (dot-product v w)
  (accumulate + 0 (map * v w)))
```

[Would of course not use lists to implement matrices in the real world — but there's still a lot that can be learned here

built-in map can take more than one sequence.

(map * '(1 2 3 4)
'(5 6 6 7))

↓

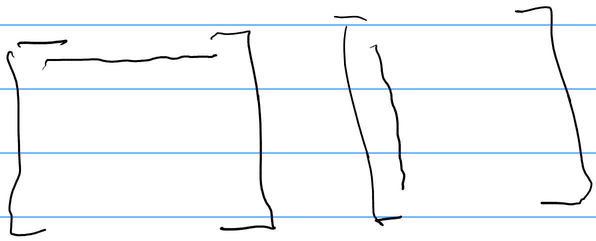
((* 1 5) (* 2 6) (* 3 6) (* 4 7))

ie

(5   12   18   28)

These are then added, via accumulate +, to deliver the dot product.

What about matrix multiplication?



m   n

We see that dot products are involved; but also that the natural computation of the ij-entry in the product is given as dot product of ith row (easy to extract from this rep of matrices) and the jth column (not so easy to extract)

One way to make columns easily extractable from matrix n would be compute the transpose of n — in which each column becomes a row.

```
(define (transpose mat)
  (accumulate-n cons '() mat))
```

discuss this in a moment

let's assume for the moment that this really does compute the transpose

```
(define (matrix-*-matrix m n)
  (let ((cols (transpose n)))
    (map (lambda (row)
       (matrix-*-vector cols row))
     m)))
```

also needs to be developed yet

As usual, when confronted with a program that someone else has written, the way to clear the fog is to use an example to develop pre/post conditions and an IH

```
(define m '((1 2 3 4) (4 5 6 6) (6 7 8 9)))

(define n '((1 2) (3 3) (4 5) (6 7)))
```

We want to examine multiplying the transpose of n by a row of m

$$\begin{pmatrix} 1 & 3 & 4 & 6 \\ 2 & 3 & 5 & 7 \end{pmatrix} \begin{pmatrix} 4 \\ 5 \\ 6 \\ 6 \end{pmatrix}$$

Transpose of n

a row of m

$$\underset{m}{\begin{pmatrix} 1 & 2 & 3 & 4 \\ 4 & 5 & 6 & 6 \\ 6 & 7 & 8 & 9 \end{pmatrix}} \underset{n}{\begin{pmatrix} 1 & 2 \\ 3 & 3 \\ 4 & 5 \\ 6 & 7 \end{pmatrix}}$$

$\underset{3\times 4}{m} \times \underset{4\times 2}{n}$ is 3×2

The (2 1) entry is (dot-product '(4 5 6 6) '(1 3 4 6))

dot products are commutative — so (dot-product '(1 3 4 6) '(4 5 6 6)) =

(dot-product '(4 5 6 6) '(1 3 4 6))

use this example to work out pre/post descriptions of the code, and then understand/prove that it does what they claim.

What about accumulate-n?

A&S 2.36

;.accumulate-n, which takes as its third argument a sequence of
; sequences, all assumed to have the same number of elements

```
(define (accumulate-n op init seqs)
  (if (null? (car seqs))
    '()
    (cons (accumulate op init (map car seqs))
        (accumulate-n op init (map cdr seqs)))))
```

in LISP, this is mapcar

in LISP, this is mapcdr

No time, I fear, to talk about recursion in TLS via the Y-combinator.

I've posted an abbreviated intro to this topic on Teams.

Maybe look at it if your July 4^{th} party gets slow...