Algorithms - Spring '25

Backtracking:
games
subset sum
text splitting

Recap

- HW1: due
- HW2: over backtracking
  due
- Readings posted

Ch 2: Backtracking:

Many of you saw in AI, apparently!
(Don't worry if not...)

Why we discuss:
It's really recursion ~~a~~ (again)!

Also really a form of brute force:
try everything recursively, & see what works.

↳ dyn. programming

N Queens

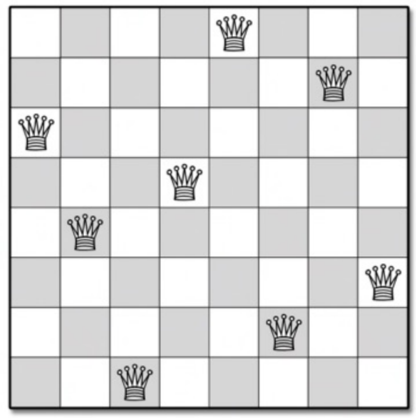


Figure 2.1. Gauss's first solution to the 8 queens problem, represented by the array [5, 7, 1, 4, 2, 8, 6, 3]

Issue: representation!

His choice: one per row, so store index of queen on rows in array.

Now, how to solve:

brute force! Place a queen + keep going.

If you get stuck, "unplace" last queen + back up.

The tree (b/c pretty) :)

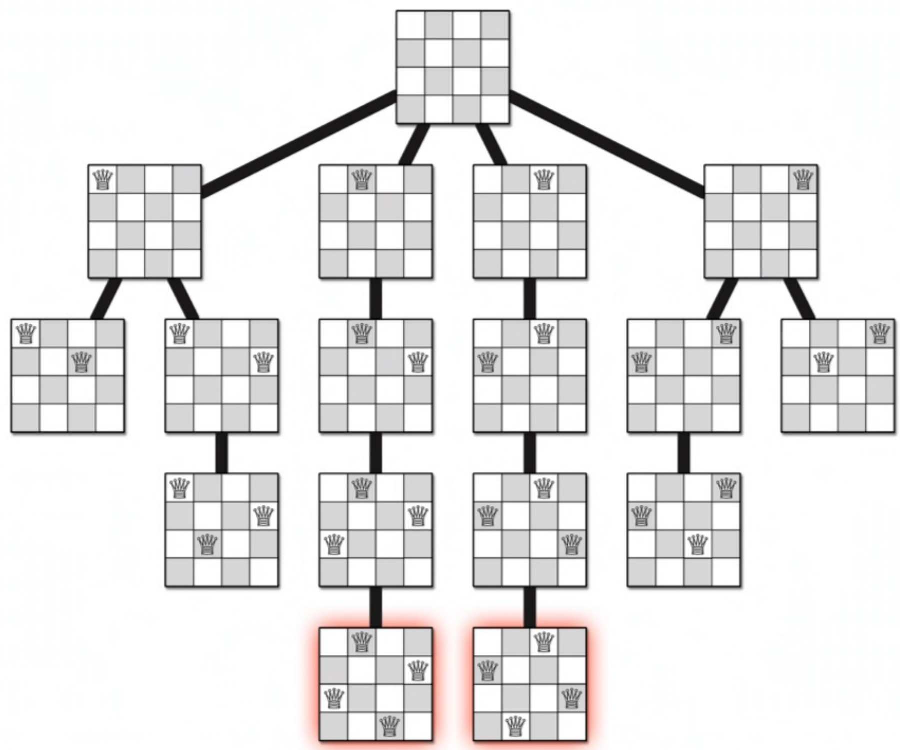


Figure 2.3. The complete recursion tree of Gauss and Laquière's algorithm for the 4 queens problem.

Problem (& hard part):

Formalizing this in code.

Sketch:

Result:

```
PLACEQUEENS(Q[1..n], r):
  if r = n+1
      print Q[1..n]
  else
      for j ← 1 to n
          legal ← TRUE
          for i ← 1 to r−1
              if (Q[i] = j) or (Q[i] = j+r−i) or (Q[i] = j−r+i)
                  legal ← FALSE
          if legal
              Q[r] ← j
              PLACEQUEENS(Q[1..n], r+1)      ⟨⟨Recursion!⟩⟩
```

Figure 2.2. Gauss and Laquière's backtracking algorithm for the n queens problem.

Runtime:

$Q(n) =$

Game Trees:

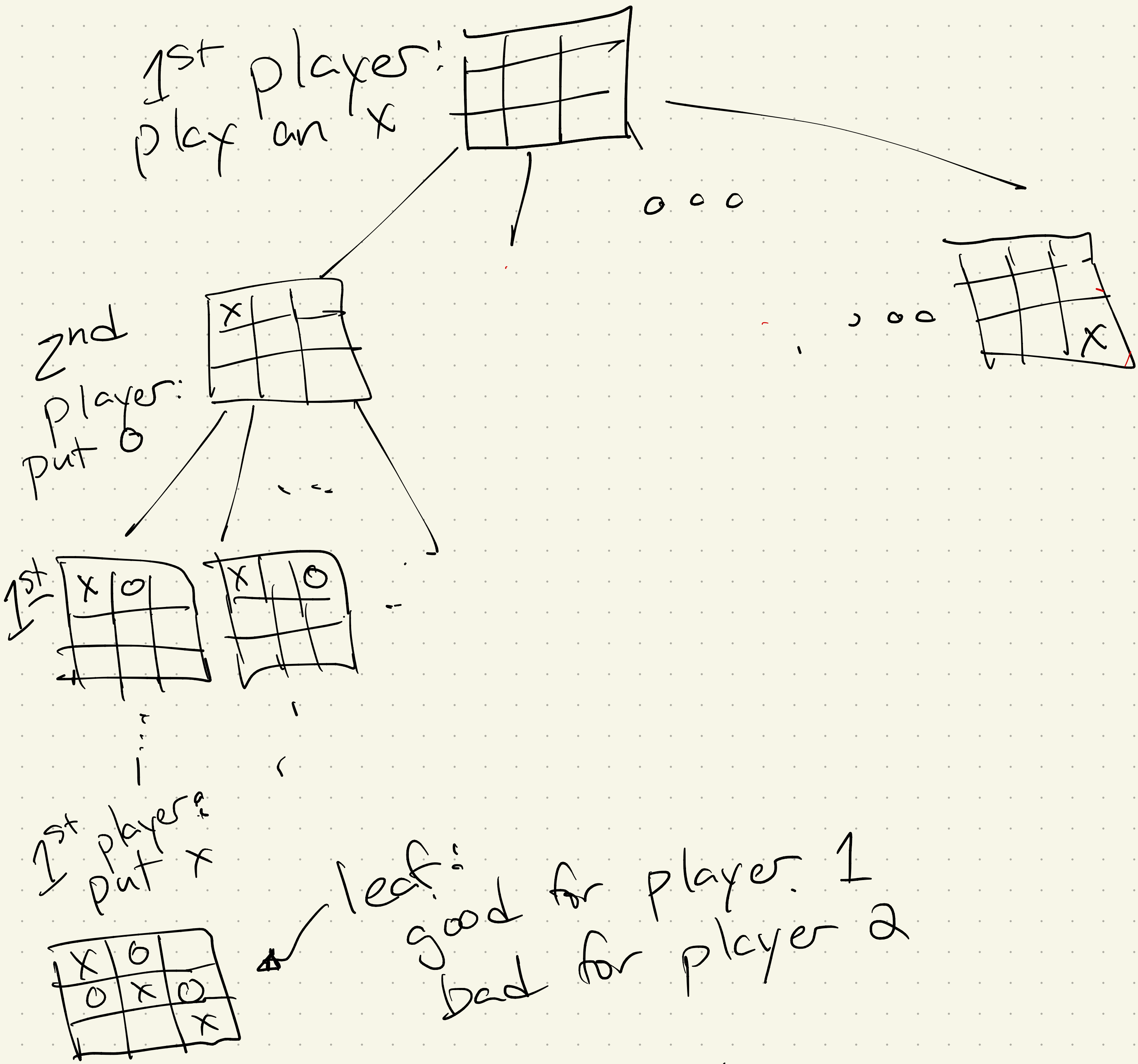
a way to model moves in 2-player games

Assume:

- No randomness so the game is just 2 people taking turns
  Ex: Checkers, chess, Nim, Go
  (not settlers of Catan!)
- "Perfect" players:
  Makes rational decisions, + if there IS a move to get them to a win state, they do it!

Idea: Track current state of the game, as play occurs

Tic-tac-toe:



Model every possible move.

A state is <u>good</u> for player 1 if they either have won, or could move to a bad state for player 2.

and bad if they have lost, or if all possible moves lead to a state that is good for player 2.

Think from the bottom up:

Tic-tac-to again:

2's turn

X	O	X
O	X	

good or bad?

1's turn

X	O	X
O	X	O

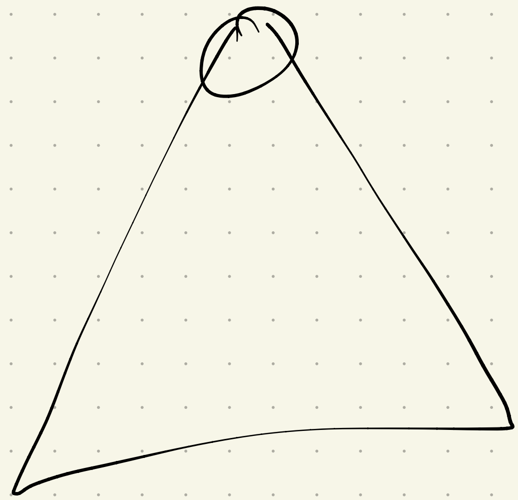
This is good for 1. (He can move somewhere bad for 2)

X	O	X
O	X	O
		X

good for 1
bad for 2

So:

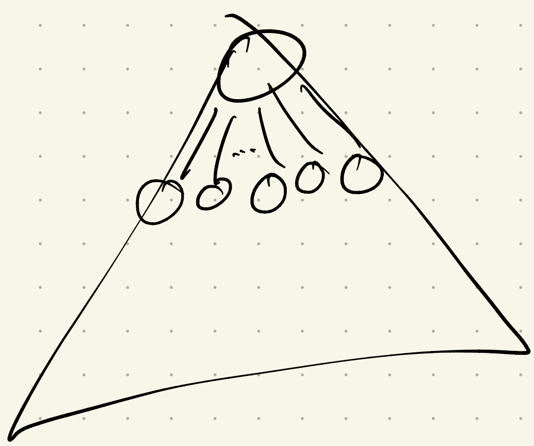
good:
I have a child who other guy thinks is bad:



Result:

Bad

All of these are good for other guy



Result:

Downsides:

Game trees are HUGE!

Tic-tac-to: over 200,000 leaves.

People can still "predict": we're good at inferring state/strategy intuitely, with practive

Computers have to search.

Hence - took 60 years to get a decent computer chess player! Need "heuristics" (aka guesses) to make it work.

Game theory — a bit more complicated.

Here, we assume clear win vs. lose.

Game theory suggests more subtle possibilites, as well as simulteneous moves & "randomness".

Example: Odds and Evens

Consider the simple game called **odds and evens**. Suppose that player 1 takes evens and player 2 takes odds. Then, each player simultaneously shows either one finger or two fingers. If the number of fingers matches, then the result is *even*, and player 1 wins the bet ($2). If the number of fingers does not match, then the result is *odd*, and player 2 wins the bet ($2). Eacl player has two possible strategies: show one finger or show two fingers. The *payoff matrix* shown below represents the payof to player 1.

Payoff Matrix

Strategy		Player 2: 1	Player 2: 2
Player 1	1	2	-2
	2	-2	2

Even if both know outcomes, result is unclear!

Example: Subset Sum

Given a set X of positive integers and a target value t, is there a subset of X which sums to t?

Ex: $X = \{8, 6, 7, 3, 10, 5, 9\}$

$t = 15$

How would we solve?

Consider one at a time:

$$X = \{8, 6, 7, 5, 3, 1, 9\}$$

Formalize this: recursion!

- base case?

Algorithm:

reset to use arrays.

⟨⟨*Does any subset of X sum to T?*⟩⟩
<u>SUBSETSUM(X, T):</u>
```
if T = 0
    return TRUE
else if T < 0 or X = ∅
    return FALSE
else
    x ← any element of X
    with ← SUBSETSUM(X \ {x}, T − x)    ⟨⟨Recurse!⟩⟩
    wout ← SUBSETSUM(X \ {x}, T)        ⟨⟨Recurse!⟩⟩
    return (with ∨ wout)
```

⟨⟨*Does any subset of $X[1..i]$ sum to T?*⟩⟩
<u>SUBSETSUM(X, i, T):</u>
```
if T = 0
    return TRUE
else if T < 0 or i = 0
    return FALSE
else
    with ← SUBSETSUM(X, i − 1, T − X[i])    ⟨⟨Recurse!⟩⟩
    wout ← SUBSETSUM(X, i − 1, T)           ⟨⟨Recurse!⟩⟩
    return (with ∨ wout)
```

<u>Correctness</u>: inductive proof, on size of X, i

<u>Base cases</u>:

$i = |X| = 0$ (so $X = \{\}$):

Ind Hyp: works for $X[1..n-1]$
or smaller values of T

Ind step: Full array $X[1..n]$

Consider $X[n]$:

Runtime:

Text Segmentation

Fix a "language", so can recognize "words".

Ex: - English text

- palindromes

- genetic data

⇒ Subroutine IsWord(s) will be given

Q: What happens to a smaller word that overlaps or is later?

Ex:

BLUE	STEM	UNIT	ROBOT	HEARTHANDSATURNSPIN
BLUEST	EMU	NITRO	BOT	HEARTHANDSATURNSPIN

Code:

```
SPLITTABLE(A[1..n]):
  if n = 0
      return TRUE
  for i ← 1 to n
      if ISWORD(A[1..i])
          if SPLITTABLE(A[i+1..n])
              return TRUE
  return FALSE
```

Runtime:

Issue w/ passing arrays:

His solution: (language independent!)

Passing by index / ptr / global / etc:

Given an index i, find a segmentation of the suffix $A[i..n]$.

Formalize an (ugly?) recursion:

$$Splittable(i) = \begin{cases} \text{TRUE} & \text{if } i > n \\ \bigvee_{j=i}^{n} \big(\text{IsWord}(i,j) \land Splittable(j+1)\big) & \text{otherwise} \end{cases}$$

& then code it:

```
⟨⟨Is the suffix A[i..n] Splittable?⟩⟩
SPLITTABLE(i):
  if i > n
      return TRUE
  for j ← i to n
      if IsWORD(i, j)
          if SPLITTABLE(j + 1)
              return TRUE
  return FALSE
```

Note: this is harder than it looks!!