Last time: Recursion Trees

$$T(n) = rT\left(\frac{n}{c}\right) + f(n)$$

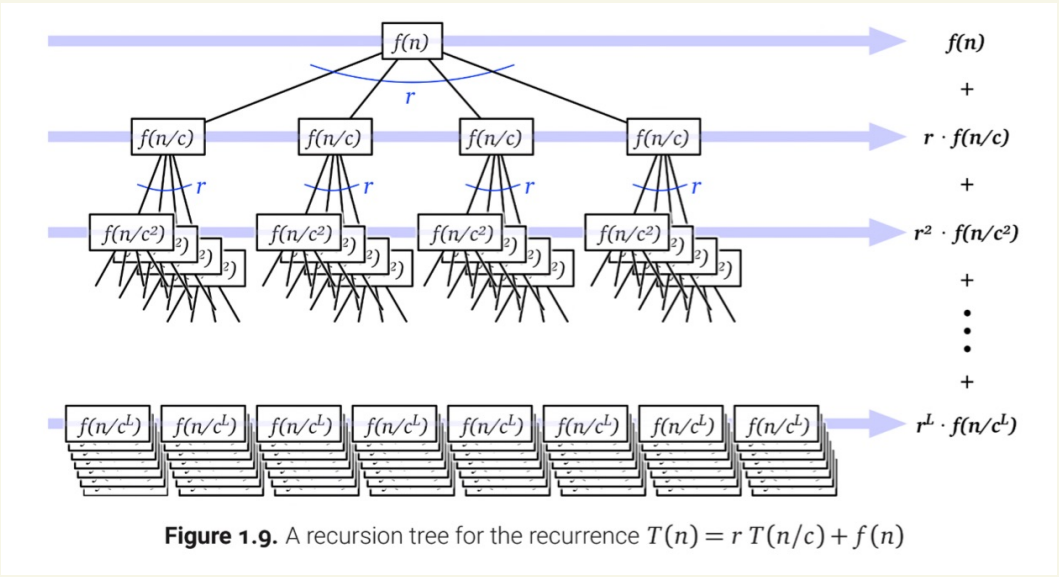


Figure 1.9. A recursion tree for the recurrence $T(n) = r\,T(n/c) + f(n)$

To solve: Sum all work in the tree!

Master Theorem

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \Rightarrow$$

$$T(n) = \begin{cases} \Theta\left(n^{\log_b a}\right) & f(n) = O\left(n^{\log_b a - \varepsilon}\right) \\ \Theta\left(n^{\log_b a} \log n\right) & f(n) = \Theta\left(n^{\log_b a}\right) \\ \Theta(f(n)) & f(n) = \Omega\left(n^{\log_b a + \varepsilon}\right) \text{AND} \\ & af(n/b) < cf(n) \text{ forlargen} \end{cases} \quad \begin{matrix} \varepsilon > 0 \\ c < 1 \end{matrix}$$

Combining the three cases above gives us the following "master theorem".

Theorem 1 *The recurrence*

$$\begin{aligned} T(n) &= aT(n/b) + cn^k \\ T(1) &= c, \end{aligned}$$

where a, b, c, and k are all constants, solves to:

$$\begin{aligned} T(n) &\in \Theta(n^k) \text{ if } a < b^k \\ T(n) &\in \Theta(n^k \log n) \text{ if } a = b^k \\ T(n) &\in \Theta(n^{\log_b a}) \text{ if } a > b^k \end{aligned}$$

THEOREM 2 **MASTER THEOREM** Let f be an increasing function that satisfies the recurrence relation

$$f(n) = af(n/b) + cn^d$$

whenever $n = b^k$, where k is a positive integer, $a \geq 1$, b is an integer greater than 1, and c and d are real numbers with c positive and d nonnegative. Then

$$f(n) \text{ is } \begin{cases} O(n^d) & \text{if } a < b^d, \\ O(n^d \log n) & \text{if } a = b^d, \\ O(n^{\log_b a}) & \text{if } a > b^d. \end{cases}$$

Other examples

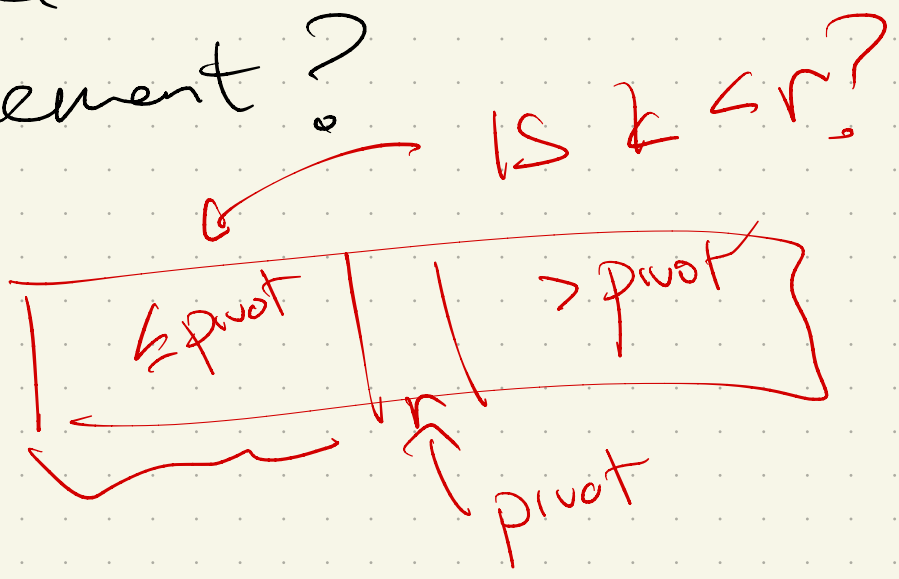
Medians: find "middle" element.

Two were covered:

```
QUICKSELECT(A[1..n], k):
  if n = 1
      return A[1]
  else
      Choose a pivot element A[p]
      r ← PARTITION(A[1..n], p)
      if k < r
          return QUICKSELECT(A[1..r−1], k)
      else if k > r
          return QUICKSELECT(A[r+1..n], k−r)
      else
          return A[r]
```

Figure 1.12. Quickselect, or one-armed quicksort

Q: How do we know which side has the k^{th} element?



Runtime

Still depends on pivot!

worst case:
choose 1st or last element

$$T(n) = (n-1) + T(n-1)$$

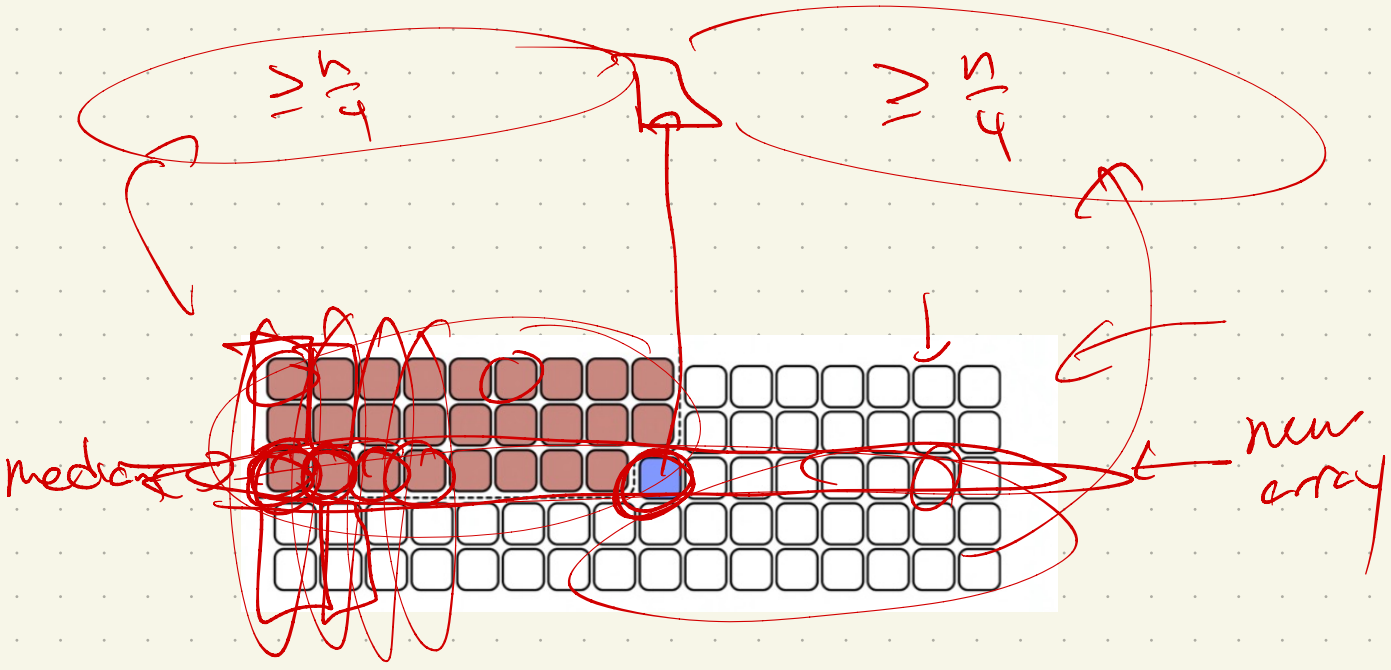
(unroll)

$$= O(n^2)$$

"Faster" version:

use for loop (still $O(1)$ time)

```
MomSelect(A[1..n], k):
  if n ≤ 25   ⟨⟨or whatever⟩⟩
      use brute force
  else
      m ← ⌈n/5⌉
      for i ← 1 to m
          M[i] ← MedianOfFive(A[5i−4..5i])   ⟨⟨Brute force!⟩⟩
      mom ← MomSelect(M[1..m], ⌊m/2⌋)        ⟨⟨Recursion!⟩⟩
      r ← Partition(A[1..n], mom)
      if k < r
          return MomSelect(A[1..r−1], k)      ⟨⟨Recursion!⟩⟩
      else if k > r
          return MomSelect(A[r+1..n], k−r)    ⟨⟨Recursion!⟩⟩
      else
          return mom
```



The recurrence:

```
MomSelect(A[1..n], k):
  if n ≤ 25   ⟨⟨or whatever⟩⟩
    use brute force
  else
    m ← ⌈n/5⌉
    for i ← 1 to m
      M[i] ← MedianOfFive(A[5i−4..5i])   ⟨⟨Brute force!⟩⟩
    mom ← MomSelect(M[1..m], ⌊m/2⌋)      ⟨⟨Recursion!⟩⟩
    r ← Partition(A[1..n], mom)
    if k < r
      return MomSelect(A[1..r−1], k)     ⟨⟨Recursion!⟩⟩
    else if k > r
      return MomSelect(A[r+1..n], k−r)   ⟨⟨Recursion!⟩⟩
    else
      return mom
```

Multiplication:

Known "fact":

$$(10^m a + b)(10^m c + d) = 10^{2m} ac + 10^m (bc + ad) + bd$$

Example:

$$102568 \times 358691 =$$

↳ Why does this suggest recursion??

The algorithm:

```
SplitMultiply(x, y, n):
  if n = 1
      return x · y
  else
      m ← ⌈n/2⌉
      a ← ⌊x/10^m⌋;  b ← x mod 10^m        ⟨⟨x = 10^m a + b⟩⟩
      c ← ⌊y/10^m⌋;  d ← y mod 10^m        ⟨⟨y = 10^m c + d⟩⟩
      e ← SplitMultiply(a, c, m)
      f ← SplitMultiply(b, d, m)
      g ← SplitMultiply(b, c, m)
      h ← SplitMultiply(a, d, m)
      return 10^(2m) e + 10^m (g + h) + f
```

Runtime:

A better trick:

$$ac + bd - (a-b)(c-d) = bc + ad$$

```
FastMultiply(x, y, n):
  if n = 1
      return x · y
  else
      m ← ⌈n/2⌉
      a ← ⌊x/10^m⌋;  b ← x mod 10^m        ⟨⟨x = 10^m a + b⟩⟩
      c ← ⌊y/10^m⌋;  d ← y mod 10^m        ⟨⟨y = 10^m c + d⟩⟩
      e ← FastMultiply(a, c, m)
      f ← FastMultiply(b, d, m)
      g ← FastMultiply(a − b, c − d, m)
      return 10^(2m) e + 10^m (e + f − g) + f
```

Runtime:

Exponentiation:

Still open!

(Amazing, right??)

The algorithms do very well:

- to compute a^n, need $O(\log n)$ multiplications

However, doesn't achieve lowest possible for every value - it's just with a constant!

Ch 2: Backtracking:

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

Why we discuss:

It's really recursion (again)!

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

N Queens

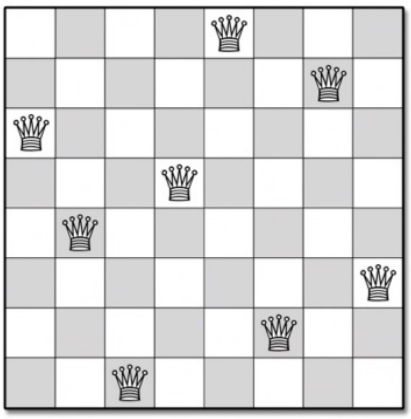


Figure 2.1. Gauss's first solution to the 8 queens problem, represented by the array [5, 7, 1, 4, 2, 8, 6, 3]

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) :

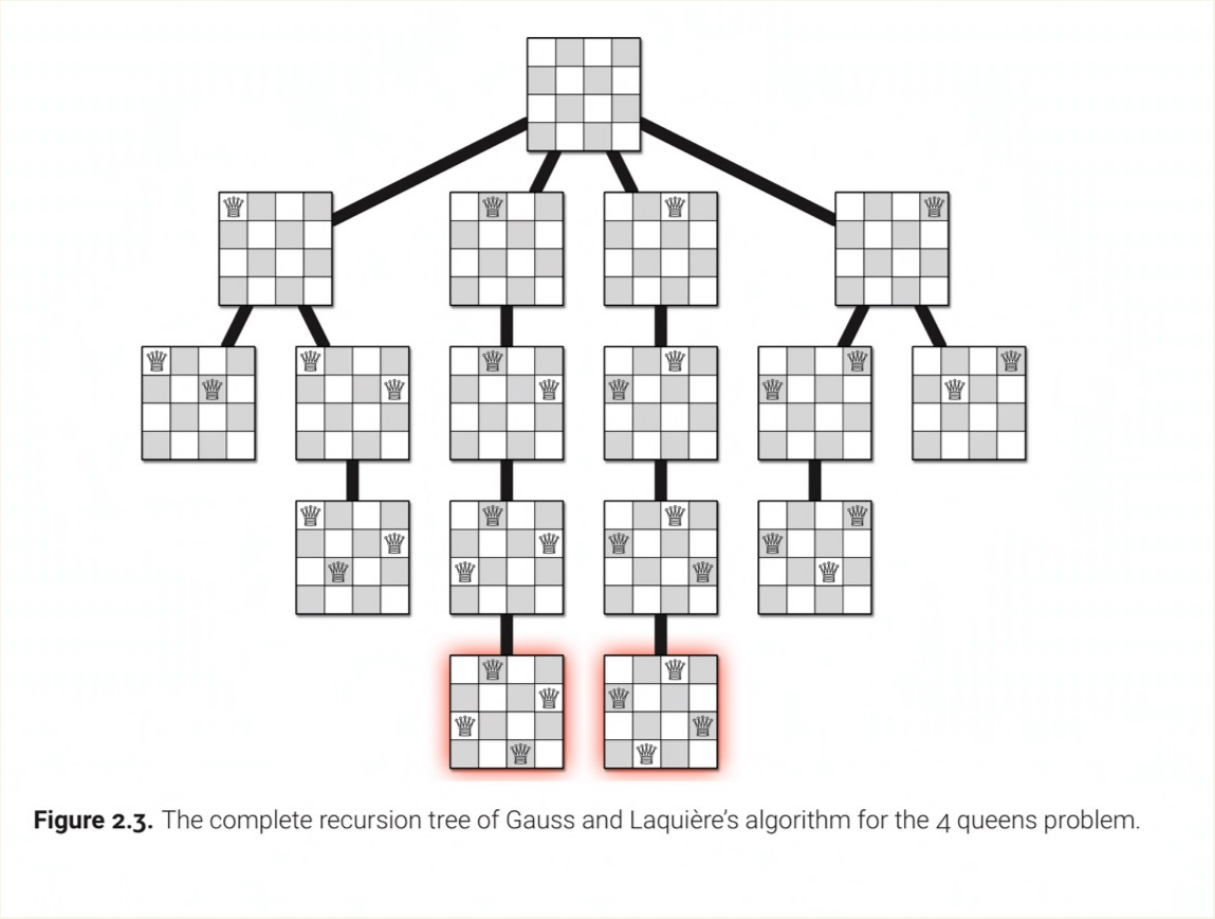


Figure 2.3. The complete recursion tree of Gauss and Laquière's algorithm for the 4 queens problem.

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:

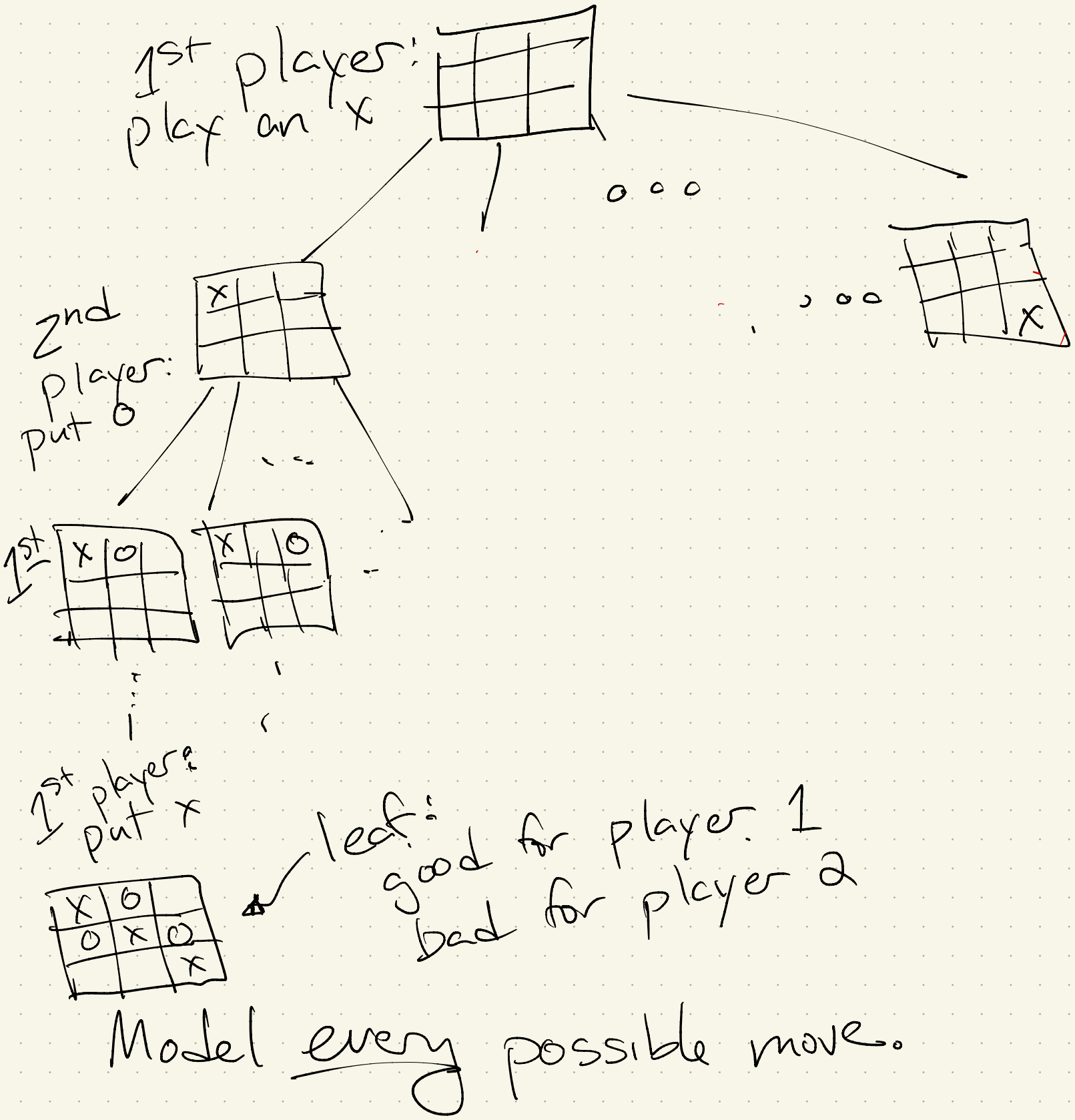
a way to model moves in 2-player games

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.

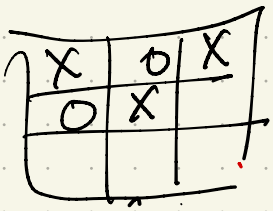
A state is good for player 1 if they either have won, or could move to a bad state for player 2.

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:

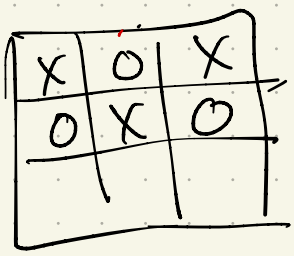
Tic-tac-to again:

2's turn

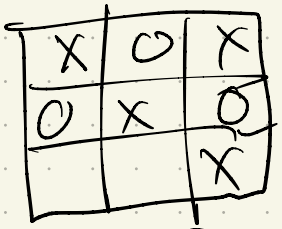


good or bad?

1's turn



↖ This is good for 1. (He can move some where bad for 2)

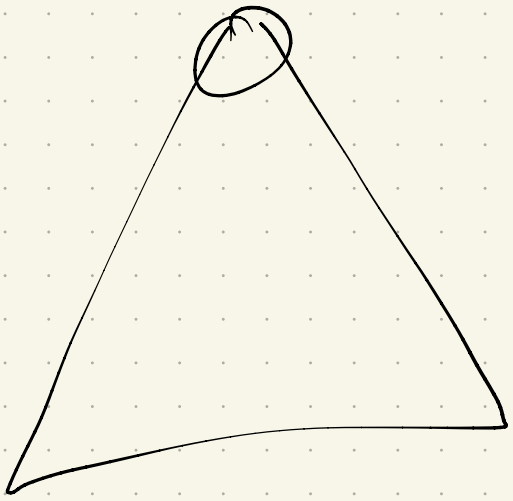


good for 1
bad for 2

So:

good:

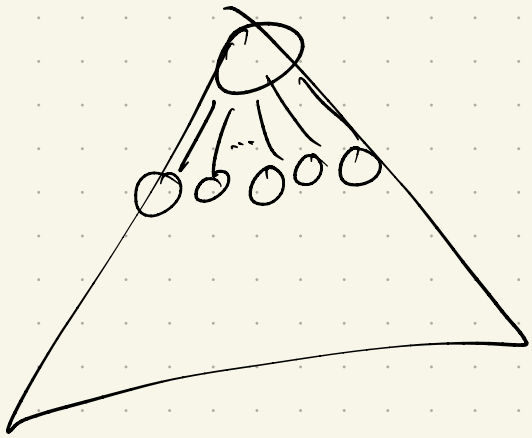
I have a child who other guy thinks is bad:



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 intuitively,
with practice

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 simultaneous 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). Each player has two possible strategies: show one finger or show two fingers. The *payoff matrix* shown below represents the payoff 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!

or <u>base case?</u>

Algorithm: reset to use arrays.

```
⟨⟨Does any subset of X sum to T?⟩⟩
SUBSETSUM(X, T):
  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?⟩⟩
SUBSETSUM(X, i, T):
  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)
```

Correctness: inductive proof, on size of X, i

Base cases:

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

2nd Hyp: works for $X[1..n-1]$ or smaller values of T

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

Consider $X[n]$:

Runtime: