Algorithms – Spring 25

Recursion
Backtracking

Recap

- HW due Friday
  - ↳ If algorithm is requested, 3 parts:
    - pseudocode (+description)
    - correctness
    - Runtime
- Teams of up to 3 (sign up on gradescope)

Last time: Recursion Trees

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

r, c constant

level 0

1

2

⋮

level i

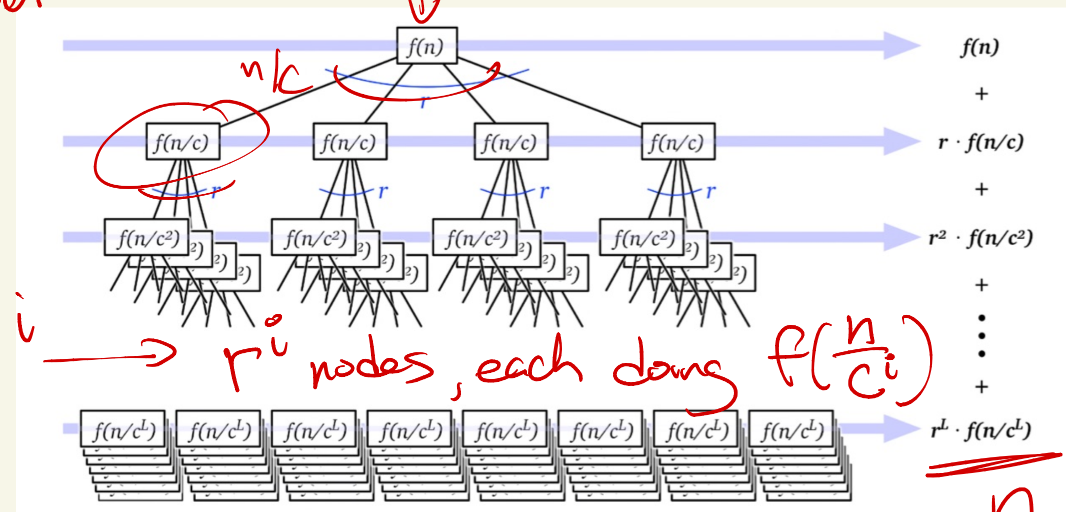


Figure 1.9. A recursion tree for the recurrence $T(n) = r\,T(n/c) + f(n)$

n/c

r^i nodes, each doing $f\left(\frac{n}{c^i}\right)$

$$\frac{n}{c^L} = 1$$

To solve: Sum all work in the tree!

$$\sum_{\text{levels } i \text{ in tree}} (\text{work on level } i)$$

is this a geom. series?

$$= \sum_{i=0}^{\text{depth}} (\#\text{nodes})(\text{work per node}) = \sum_{i=0}^{\log_c n} r^i \cdot f\left(\frac{n}{c^i}\right)$$

Master Theorem

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

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

$f(n) << n^{\log_b a}$

$f(n) == n^{\log_b a}$

$f(n) >> n^{\log_b a}$

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

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

poly

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

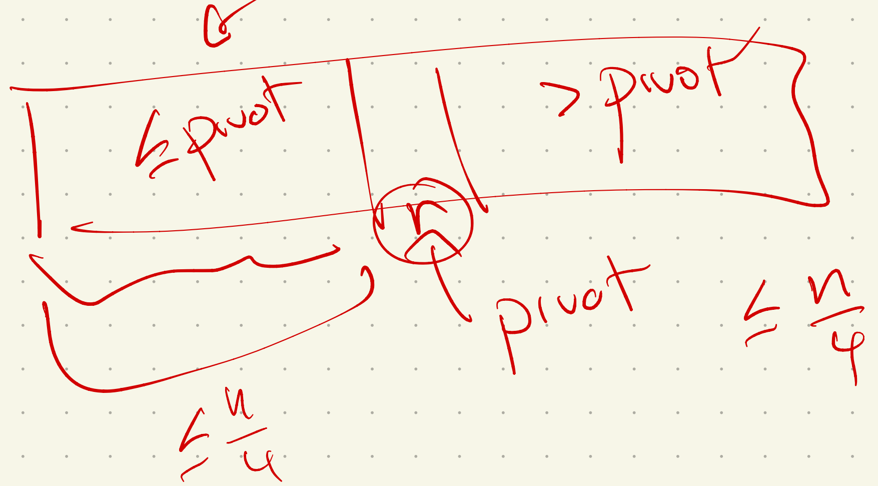
Medians: find "middle" element.
Two were covered:

```
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?

Is $k < r$?



Runtime

Still depends on pivot!

worst case:
choose 1^{st} or last
element

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

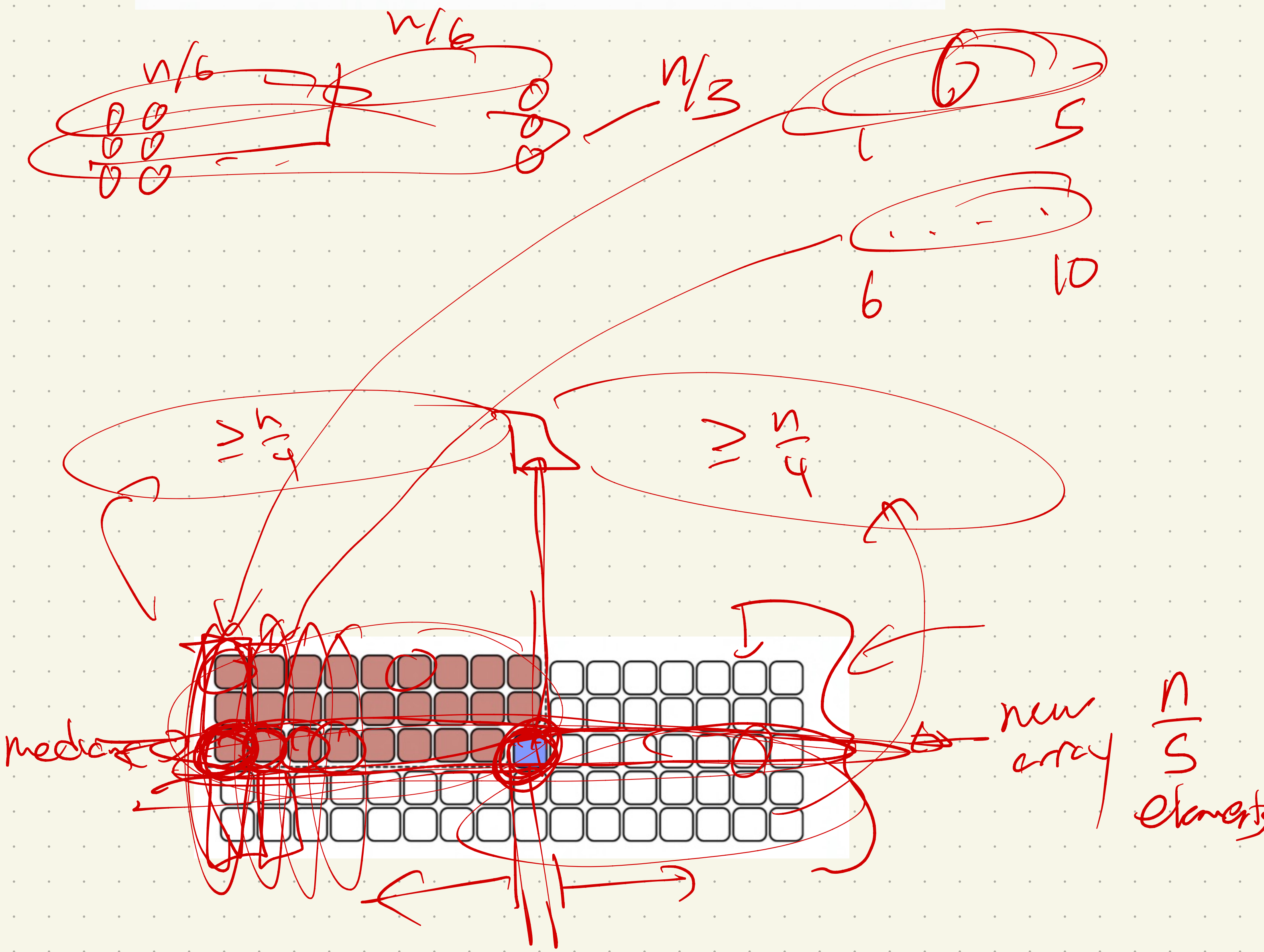
(unroll)

$$= O(n^2)$$

"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
```

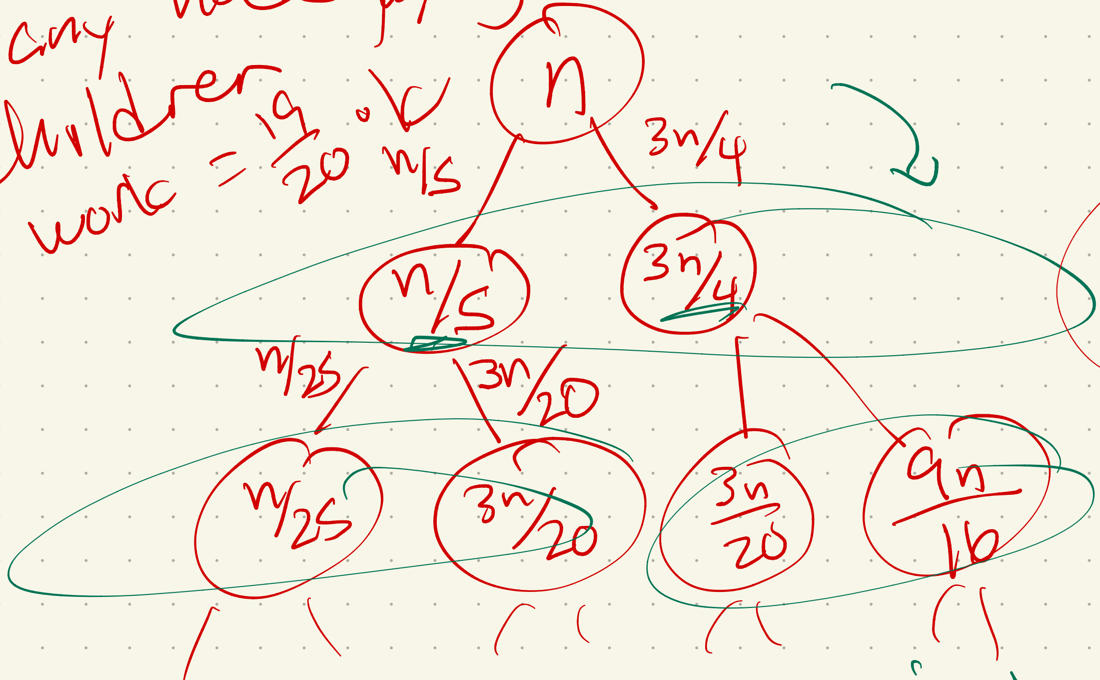
$O(n)$

$O(n)$

$T(n)$ = runtime for n elt list

$$\leq O(n) + T\left(\frac{n}{5}\right) + T\left(\frac{3n}{4}\right)$$

for any node w/ k, 2 children have work $= \frac{19}{20} \cdot k$



$\frac{19}{20} \cdot n$

$\left(\frac{19}{20}\right)^2 \cdot n$

level i: $\left(\frac{19}{20}\right)^i \cdot n$

Tree has depth

$$\log_5 n \le d \le \log_{4/3} n$$

divide by 5 (left branch)

multiply by 3/4 ↪ div by 4/3

$$T(n) = \sum_{i=0}^{\log n} (\text{work per level})$$

$r < 1$

$$= \sum_{i=0}^{\log n} \left(\frac{19}{20}\right)^i \cdot \underline{n}$$

$$\le n \sum_{i=0}^{\infty} \left(\frac{19}{20}\right)^i = n\left(\frac{1}{1-\frac{19}{20}}\right)$$

(geom series idea)

$$= \underline{O(n)}$$

Why? Recall quicksort:

$$T(n) = \max_r \{ \underbrace{T(r) + T(n-r)}_{\text{2 calls}} + \underbrace{\Theta(n)}_{\text{pivot code}} \}$$

Instead of random pivot, call median:

$$T(n) = \underbrace{O(n)}_{\text{call MoM}} + \underbrace{O(n)}_{\text{pivot}} + 2T(\tfrac{n}{2})$$

"Quicksort" in $n \log n$

Multiplication:

2m digit #

$\underbrace{a}_{m}\ \underbrace{b}_{m}$

Known "fact":

$$(10^m a + b)(10^m c + d) =$$

$$10^{2m}ac + 10^m(bc + ad) + bd$$

Example:

$$102568 \times 358691$$

$$= (\underbrace{102}_{a} \times 10^3 + \underbrace{568}_{b}) \times (\underbrace{358}_{c} \times 10^3 + \underbrace{691}_{d})$$

$$=$$

↳ Why does this suggest recursion??

^ multiply smaller #s!

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)   ← a·c
    f ← SPLITMULTIPLY(b, d, m)   ← b·d
    g ← SPLITMULTIPLY(b, c, m)   ← b·c
    h ← SPLITMULTIPLY(a, d, m)   ← a·d
    return 10^{2m} e + 10^m (g + h) + f
```

additions + 0-padding

Runtime:

$$T(m) = O(1) + 4T\left(\frac{m}{2}\right)$$

Master thm: $f(n) = O(1) = n^0$

$d = 0$

$a = 4$

$b = 2$

$a = 4 > b^d = 2^0 = 1$

$$T(m) = m^{\log_2 4} = m^2$$

A better trick

$$ac - bc + bd - ad$$

$$4\left[\underbrace{ac}_{e} + \underbrace{bd}_{f} - (a-b)(c-d) = bc + ad\right]$$

Why?

```
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
```

$$(e+f-g) = bc+ad$$

g

Runtime:

$$T(n) = O(1) + 3T\left(\frac{n}{2}\right)$$

n^0, so $d=0$ $\quad a=3 \quad b=2$

MT: $3 > 2^0$

$$T(n) = \Theta\left(n^{\log_2 3}\right)$$

(addition or bit tricks)

$$1 < \log_2 3 < 2$$

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.

↳ dyn. programming

N Queens

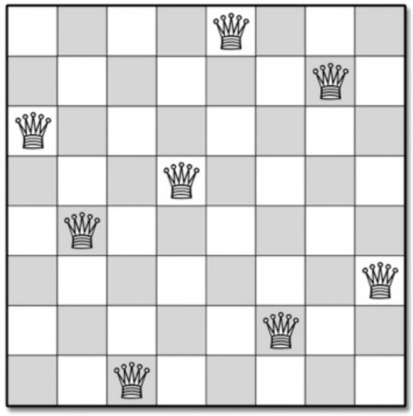


Figure 2.1. Gauss's first solution to the 8 queens problem, represented by the array [5, 7, 1, 4, 2, 8, 6, 3]

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) ◡̈

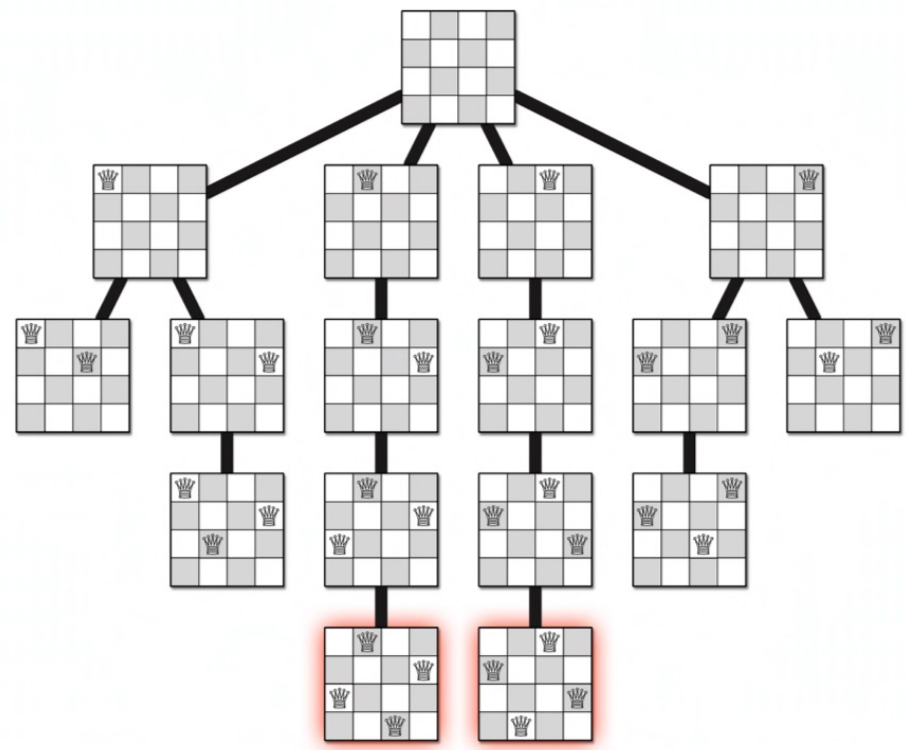


Figure 2.3. The complete recursion tree of Gauss and Laquière's algorithm for the 4 queens problem.

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:

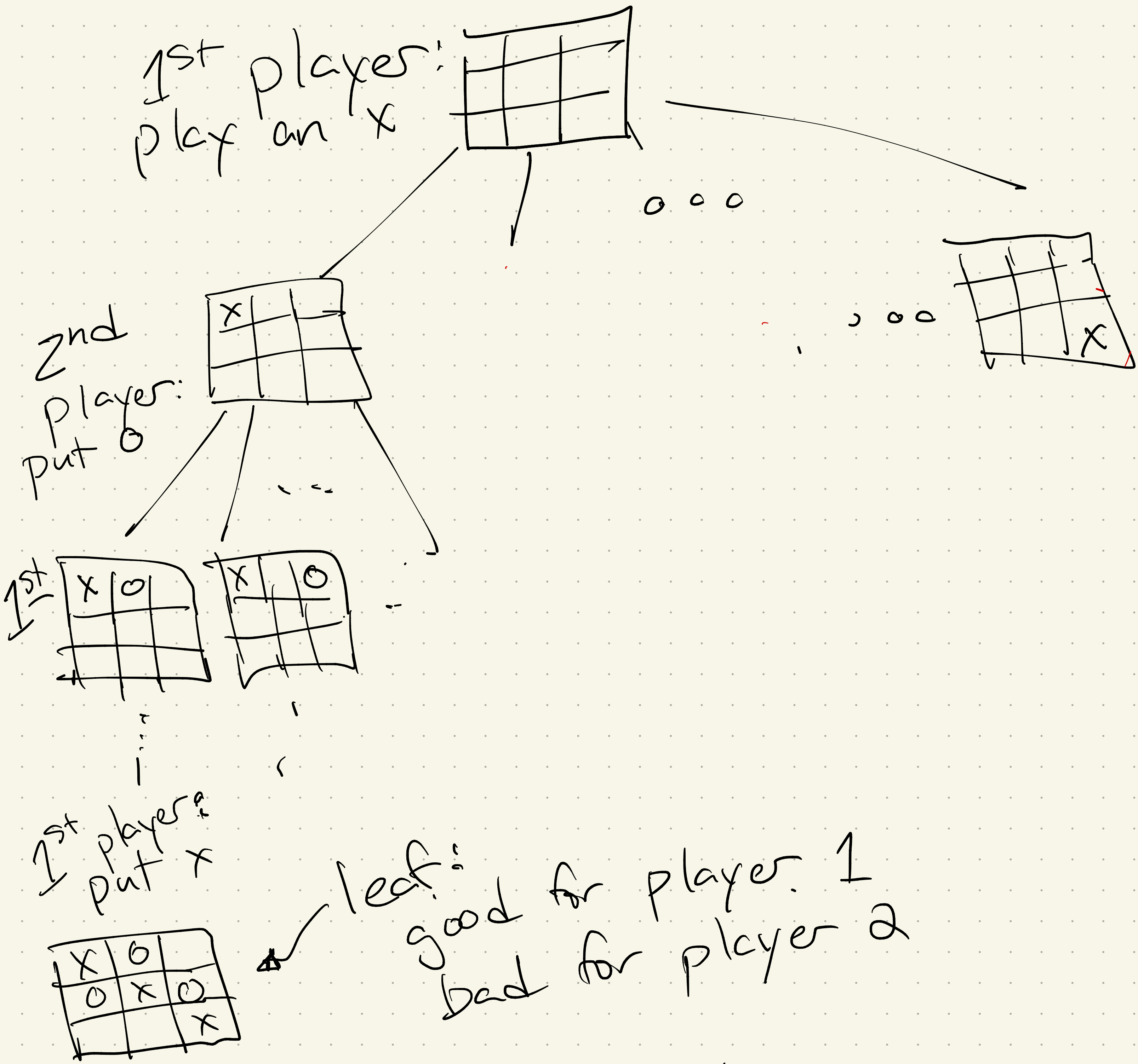
a way to model moves in 2-player games

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 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

X	O	X
O	X	

good or bad?

1's turn

X	O	X
O	X	O

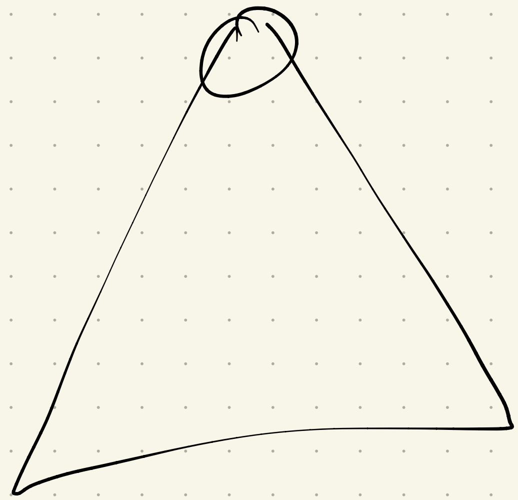
This is good for 1. (He can move some where bad for 2)

X	O	X
O	X	O
		X

good for 1
bad for 2

So:

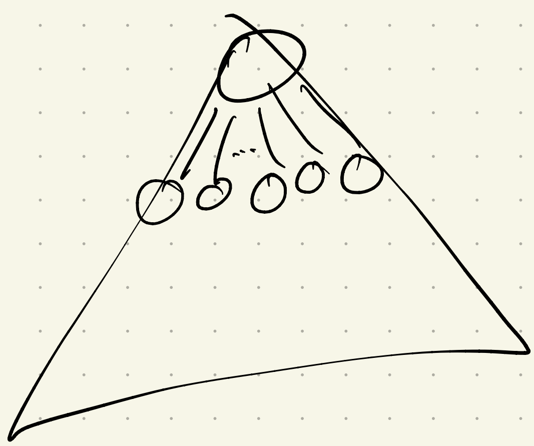
good:
I have a child who other guy thinks is bad:



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). Eac player has two possible strategies: show one finger or show two fingers. The *payoff matrix* shown below represents the payo 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 base case?

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)
```

<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: