AI-m of the Game

Workshop #3: Optimizations

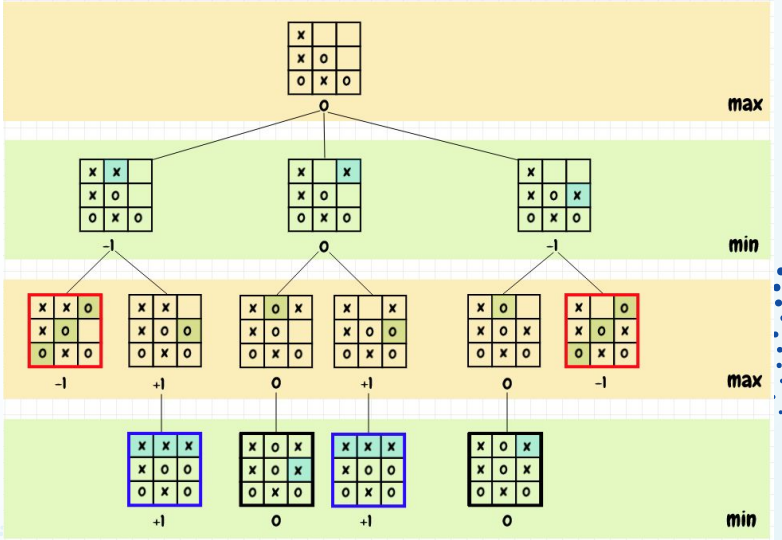
Project Ignite Instructors:
Brandon Wang, Brandon Wei

Recap

What happened last time?

Minimax Algorithm

- you try to ***maximize*** your utility
- opponent tries to ***minimize*** your utility (ie. maximize his own)
- draw a full game tree, and evaluate every state based on how your opponent will minimize and how you'll maximize
  - for now, we only care about the ***end states***



Alpha-Beta Pruning

- alpha (α) = best value for Player 1 (maximizer)
- beta (β) = best value for Player 2 (minimizer)
- alpha pruning: stop searching a MIN node if the new β value becomes ≤ the old α value
- beta pruning: stop searching a MAX node if the new α value becomes ≥ the old β value
- for a better description: see this website

Solve Tic-Tac-Toe

Tic-Tac-Toe is more complex than you think!

- 5,478 unique legal states and 255,168 games
- game tree has 549,946 nodes

From tictactoe.py

- minimax(): recursively search the children nodes
- mmpruning(): minimax w/ alpha-beta pruning to cut down computation time
  - much faster with pruning!

Solve Sim

Sim is much more complicated than Tic-Tac-Toe

- more states, more depth, and more games

From sim.py

- mmpruning(): minimax w/ alpha-beta pruning to cut down computation time
- negamax(): just a simpler form of mmpruning()

Evaluating mmpruning

Just how good is minimax w/ pruning?

How Good is Minimax + Pruning?

- Minimax alone?
  - Gives the optimal move
  - Terrible performance! Need to search every single game node!
- Minimax + Alpha-Beta Pruning?
  - Gives the same optimal move as minimax
  - Much better performance; on average, allows us to explore twice as deep as minimax alone
- Together, they can completely solve simple games like Tic-Tac-Toe in less than 1 second!

Minimax + Pruning Examples

- tictactoe.py
  - depth=9: ≈20 s w/o pruning, ≈0.5 s w/ pruning
- sim.py
  - depth=6: ≈50 s w/o pruning, ≈0.1 s w/ pruning
  - depth=8: ≈0.8 s w/ pruning
  - depth=10: ≈6 s w/ pruning
  - depth=12: ≈40 s w/ pruning
  - depth=14: ≈150 s w/ pruning (2.5 min)
  - depth=15: ≈540 s w/ pruning (9 min!)
- SLOW!!! This is problematic for our next game...

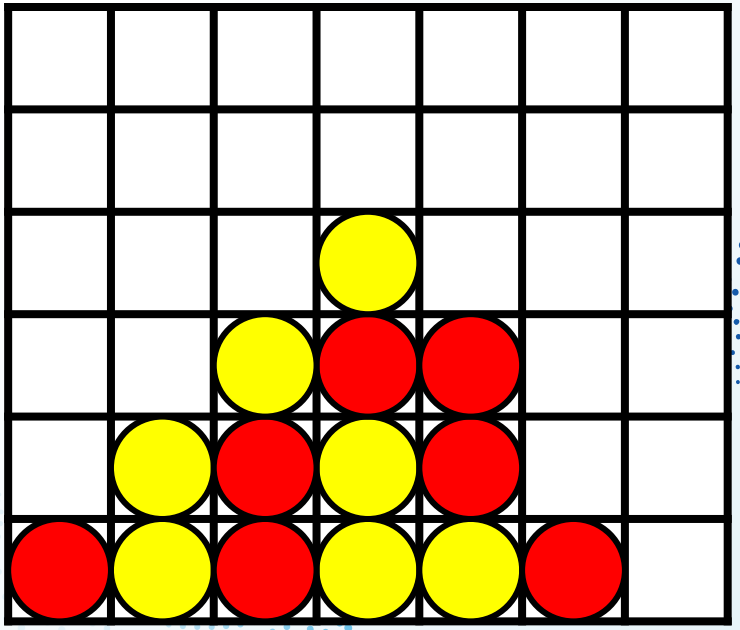
Connect Four

More complex game. 4-in-a-row, with gravity!

Rules of Connect Four

- 6 × 7 grid of squares
- Turn player places their piece on a non-filled column
  - **<u>gravity pulls it down!</u>**
  - P1: red, P2: yellow
  - The player who gets 4 of their pieces in a row, column, or diagonal wins!
- If all 42 squares are filled w/o a 4-in-a-row, it's a draw/tie

Red to move



Connect Four Complexity

- Upper limit of games: 7^{42} = 3.1E35
  - actually has 4.53E12 games
- Full game tree has 1E21 nodes
  - ≈2E15 times more complex than Tic-Tic-Toe!
- Plus, much more computation needed to evaluate the board (ie. determine if 4-in-a-row or a draw)

1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31	32	33	34	35
36	37	38	39	40	41	42

Major Changes Needed!!!

- We can’t just use minimax + pruning anymore!
  - Connect Four is much, much, much more complex and time-consuming than Tic-Tac-Toe and Sim
- What can we do to make our AI better?
  - Discuss!

Optimizations

Let’s make our AI algorithms smarter and faster!

Optimizations

- optimization = any modification to give code better ...
  - quality (ie. clearer, more concise code)
  - time efficiency (ie. computes in less time)
  - space efficiency (ie. uses up less memory space)
- related to the “Big-O Notation” (ie. the time and space usage of a function in the **worst case**)
  - the more optimal the code, the smaller the Big-O!

Optimization of divisors()

Task: count all divisors of n from 1 to n (eg. divisors(6)=4)

Solution #1: Time Complexity = O(n), Space Complexity = O(1)

```
def divisors(n):
    cnt = 0
    for i in range(1,n+1):
        if n % i == 0:
            cnt += 1
    return cnt
```

Optimization of divisors()

Task: count all divisors of n from 1 to n (eg. divisors(6)=4)

Solution #2: Time Complexity = O(n^(1/2)), Space Complexity = O(1)

```
def divisors(n):
    cnt = 0
    for i in range(1, math.ceil(math.sqrt(n))):
        if n % i == 0:
            if i*i == n:
                cnt += 1
            else:
                cnt += 2
    return cnt
```

Optimization of divisors()

Task: count all divisors of n from 1 to n (eg. divisors(6)=4)

Solution #3: Time Complexity = O(n^(1/3)), Space Complexity = O(n^2)

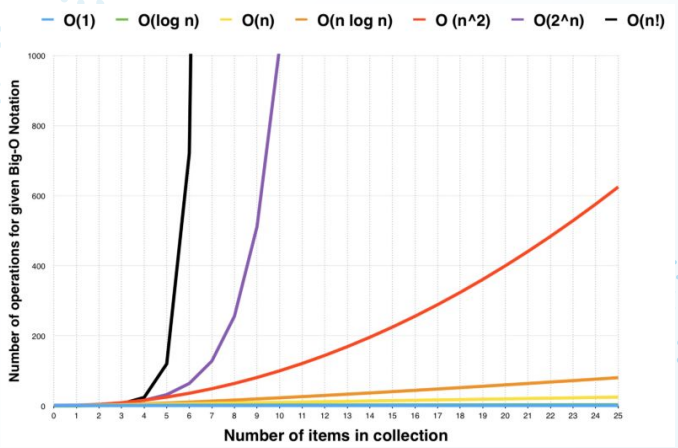
```
def SieveOfEratosthenes(n, prime, primesquare,
a):
    for i in range(2,n+1):
        prime[i] = True
    for i in range((n*n + 1)+1):
        primesquare[i] = False
    prime[1] = False
    p = 2
    while(p*p <= n):
        if (prime[p] == True):
            i = p * 2
            while(i <= n):
                prime[i] = False
                i += p
        p+=1

    j = 0
    for p in range(2,n+1):
        if (prime[p]==True):
            a[j] = p
            primesquare[p * p] = True
            j+=1
```

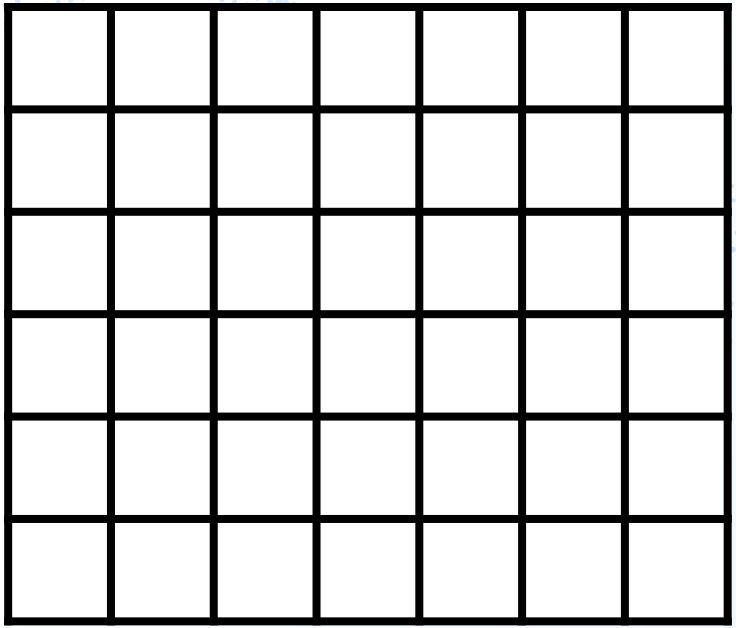
```
def divisors(n):
    if (n == 1):
        return 1
    prime = [False]*(n + 2)
    primesquare = [False]*(n*n + 2)
    a = [0]*n
    SieveOfEratosthenes(n, prime, primesquare, a)
    ans = 1
    i=0
    while(1):
        if(a[i]**3 > n):
            break
        cnt = 1
        while (n % a[i] == 0):
            n /= a[i]
            cnt += 1
        ans *= cnt
        i+=1
    n=int(n)
    if (prime[n]==True):  ans *= 2
    elif (primesquare[n]==True):  ans *= 3
    elif (n != 1):  ans *= 4
    return ans
```

Why Our AI Needs LOTS of Optimization

- More optimization, more efficient, more nodes explored in the same amount of time
  - even a little bit lets our AI explore higher depths
- Optimization is more effective the greater the number of states



Let’s Begin Optimizing!



Opt #1: Objects and Classes

Object-Oriented Programming (OOP) allows us to define objects and classes, which gives us ...

- Abstraction: a simpler way to represent code and data
  - hide the unnecessary background stuff from user
- Encapsulation: methods and fields grouped into 1 object
- Inheritance: “child” classes that adopt similar methods and fields as the “parent” classes
- Polymorphism: the same function name with different functions for different classes

Opt #1: Objects and Classes

```
class Player1:
   def welcome(self):
      print("Hi! I'm Player 1!")
   def getColor(self):
      return "red"

class Player2:
   def welcome(self):
      print("Hi! I'm Player 2!")
   def getColor(self):
      return "yellow"

P1 = Player1()
P2 = Player2()
for player in [P1, P2]:
   player.welcome()
   color = player.getColor()
```

Opt #1: Objects and Classes

```
class Player:
   def __init__(self, n, color): # a constructor to initialize values
      self.turn_order = n
      self.color = color
   def welcome(self)
      print("Hi! I'm Player %d!" % self.turn_order)
   def changeColor(color)
      self.color = color

P1 = Player(1, "red")
P2 = Player(2, "yellow")
for player in [P1, P2]:
   player.welcome()
   print("My current color is: %s" % player.color)
   player.changeColor("blue")
   print("My current color is: %s" % player.color)
```

Opt #2: Heuristics

if we reach our max depth, instead of returning 0, give a better estimate for the position

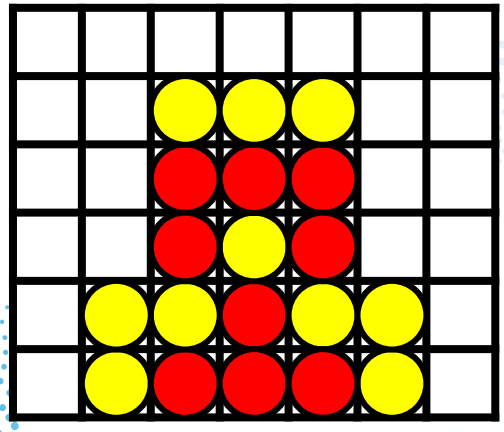
- can no longer look at future positions; use current factors
- Reasonable heuristics:
  - no. of pieces in column 4 (center)
  - for each piece: col 4 = +4, col 3/5 = +3, col 2/6 = +2, col 1/7 = +1
  - for each player, a 3-in-a-row adds +1 to their score
    heuristic = our score - opponent's score

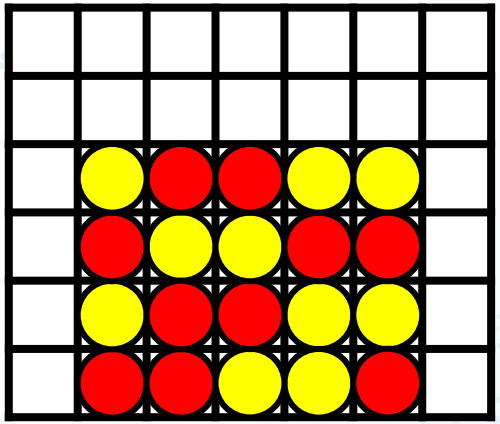
Opt #2: Heuristics

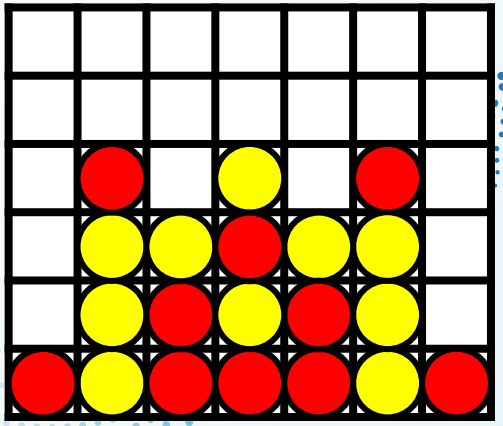
Suppose we're Player 1 (red), and we reached depth = 20

heuristic = our 3-in-a-rows – opponent's 3-in-a-rows

What's the heuristic for each board?







Opt #3: Move Exploration Order

explore BETTER moves before exploring WORSE moves

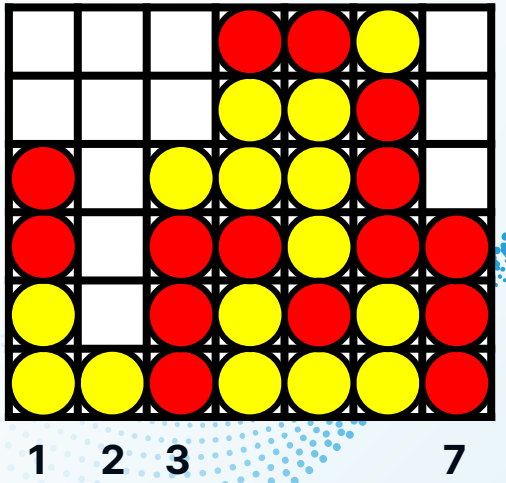
- do “winning” moves first, do “losing” moves last
- recall that in alpha-beta pruning, we prune when we already found the best move to take
  - by finding the best moves first, we can prune more!
- sim_solution.py uses this, and it makes the code run a lot faster!

Opt #3: Move Exploration Order

Suppose we're Player 1 (red)

What's the **ideal** order to explore the columns to prune the most moves?

A. 1,2,7,3
B. 2,3,7,1
C. 3,7,1,2
D. 7,3,1,2



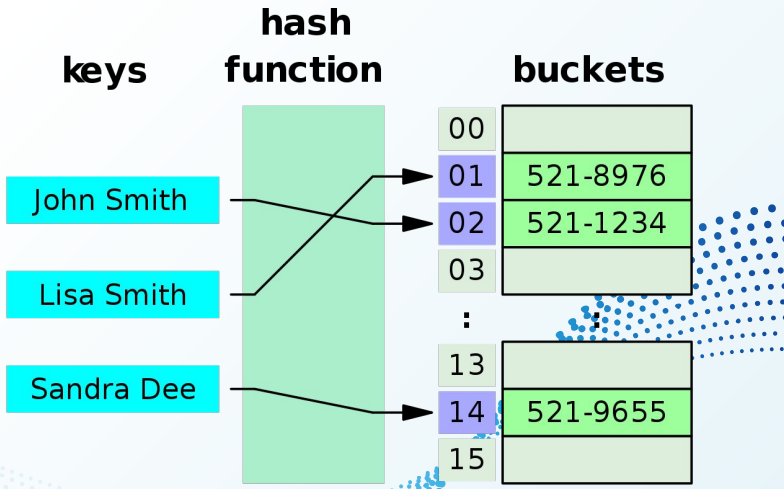
Break Point!

We’re halfway done with optimizations!

Opt #4: Memoization

Transposition Table: store the states in a large hash table we can easily call back later.

- hash table = a giant table/array “buckets”/”slots”, that maps keys to values/objects
  - choose a key: object → key
  - hash function: key → index
  - search: index → value
- very, very fast and efficient!



Opt #4: Transposition Table

Task: store people's ID #s into an ***array***

Let's choose the first empty space in the array to store the names and values

- Albert: ID=100
- Brad: ID=200
- Carl: ID=300
- Daniel: ID=400
- Ellie: ID=500
- ...

Albert ID=100	Brad ID=200	Carl ID=300	Daniel ID=400	Ellie ID=500
...	...	...	...	...

Opt #4: Transposition Table

Finding the IDs again after storing them is long and time-consuming
Time Complexity O(n)

- imagine searching through hundreds of thousands of names!

- Albert: search 1 elements
- Brad: search 2 elements
- Carl: search 3 elements
- Daniel: search 4 elements
- Ellie: search 5 elements
- ...

Albert ID=100	Brad ID=200	Carl ID=300	Daniel ID=400	Ellie ID=500
...	...	...	...	...

Opt #4: Transposition Table

Task: store people's ID #s into a ***hash table***, using first name as key
Hash table size: 10 (indices are from 0 to 9)
Hash function: (sum of letters→numbers in name) mod 10

- Albert: ID=100
- Brad: ID=200
- Carl: ID=300
- Daniel: ID=400
- Ellie: ID=500
- …

0	1	2	3	4
5	6	7	8	9

Opt #4: Transposition Table

Task: store people's ID #s into a hash table, using first name as key
Hash table size: 10 (indices are from 0 to 9)
Hash function: (sum of letters→numbers in name) mod 10

- **Albert: 1+12+2+5+18+20=58 mod 10 = 8; ID=100**
- Brad: ID=200
- Carl: ID=300
- Daniel: ID=400
- Ellie: ID=500

Opt #4: Transposition Table

Task: store people's ID #s into a hash table, using first name as key
Hash table size: 10 (indices are from 0 to 9)
Hash function: (sum of letters→numbers in name) mod 10

- Albert: 1+12+2+5+18+20=58 mod 10 = 8; ID=100
- **Brad: 2+18+1+4=25 mod 10 = 5; ID=200**
- Carl: ID=300
- Daniel: ID=400
- Ellie: ID=500

200			**100**	

Opt #4: Transposition Table

Task: store people's ID #s into a hash table, using first name as key
Hash table size: 10 (indices are from 0 to 9)
Hash function: (sum of letters→numbers in name) mod 10

- Albert: 1+12+2+5+18+20=58 mod 10 = 8; ID=100
- Brad: 2+18+1+4=25 mod 10 = 5; ID=200
- **Carl: 3+1+18+12=34 mod 10 = 4; ID=300**
- Daniel: ID=400
- Ellie: ID=500

				300
200			**100**	

Opt #4: Transposition Table

Task: store people's ID #s into a hash table, using first name as key
Hash table size: 10 (indices are from 0 to 9)
Hash function: (sum of letters→numbers in name) mod 10

- Albert: 1+12+2+5+18+20=58 mod 10 = 8; ID=100
- Brad: 2+18+1+4=25 mod 10 = 5; ID=200
- Carl: 3+1+18+12=34 mod 10 = 4; ID=300
- **Daniel: 4+1+14+9+5+12=45 mod 10 = 5; ID=400**
- Ellie: ID=500

				300
200 **400**			**100**	

COLLISION!!

1. **keep both entries in table (as a linked list)**
2. **increase size of table**

Opt #4: Transposition Table

Task: store people's ID #s into a hash table, using first name as key
Hash table size: 10 (indices are from 0 to 9)
Hash function: (sum of letters→numbers in name) mod 10

- Albert: 1+12+2+5+18+20=58 mod 10 = 8; ID=100
- Brad: 2+18+1+4=25 mod 10 = 5; ID=200
- Carl: 3+1+18+12=34 mod 10 = 4; ID=300
- Daniel: 4+1+14+9+5+12=45 mod 10 = 5; ID=400
- **Ellie: 5+12+12+9+5=43 mod 10 = 3; ID=500**

			500	**300**
200 **400**			**100**	

Opt #4: Transposition Table

Finding the IDs afterwards from the table is just as easy!
Time Complexity: O(1)

- go directly to our “bucket”; no need to search through anyone else’s
- exception: collisions

- Albert: 1+12+2+5+18+20=58 mod 10 = 8
- Brad: 2+18+1+4=25 mod 10 = 5
- Carl: 3+1+18+12=34 mod 10 = 4
- Daniel: 4+1+14+9+5+12=45 mod 10 = 5
- Ellie: 5+12+12+9+5=43 mod 10 = 3

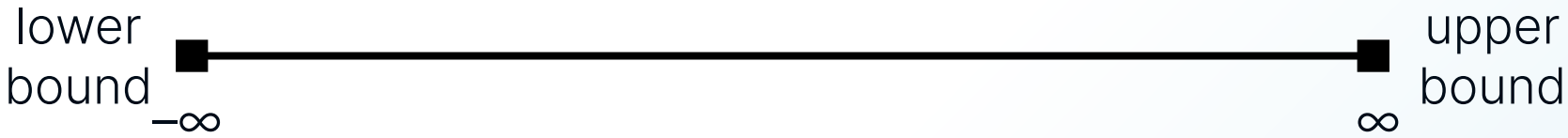
			500	**300**
200 **400**			**100**	

Opt #5: Alpha-Beta Windows

logically narrow down the interval (“window”) where the minimax value lies during alpha-beta pruning

- normal alpha-beta pruning window is [α, β]
  - start w/ α=-∞ and β=∞
- Zero/Null window: [α, α+1] (size is 0!)
  - the smaller the window, the faster the search
  - this’ll tell us if the true value is more than or less than the input guess α
  - repeatedly call this to narrow down true value

Opt #5: Alpha-Beta Windows

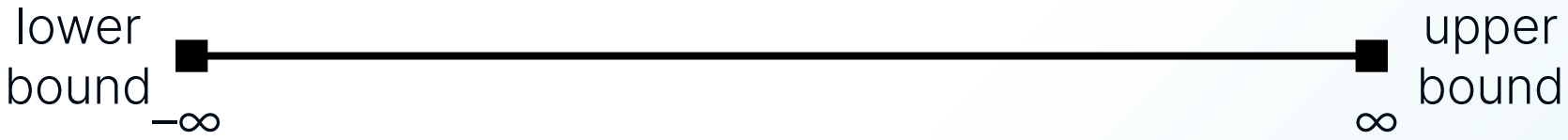


initial guess: α=0 ⟶ alpha-beta result: -5

result < guess: failed low

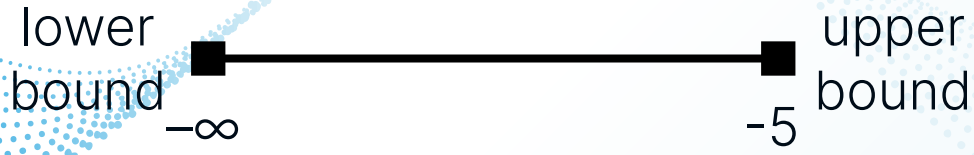
What's our new bounds?

Opt #5: Alpha-Beta Windows

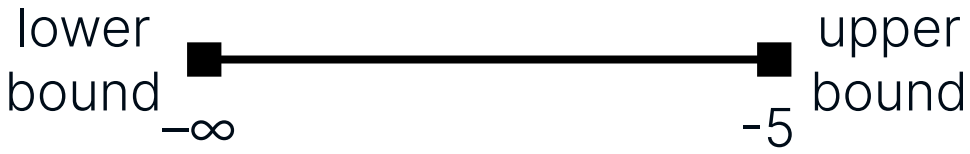


initial guess: α=0 ⟶ alpha-beta result: -5

result < guess: failed low



Opt #5: Alpha-Beta Windows

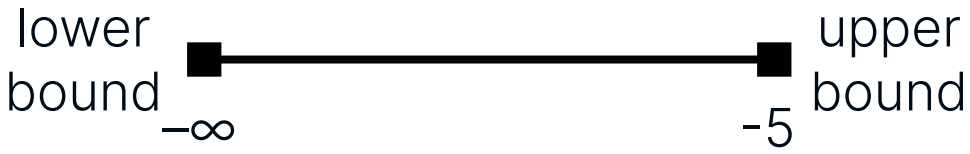


initial guess: α=-15 ⟶ alpha-beta result: -10

result > guess: failed high

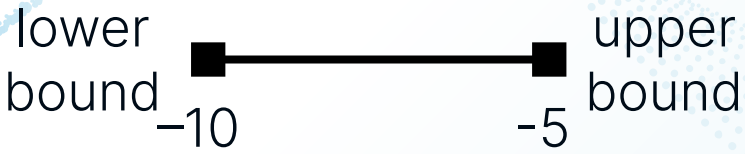
What's our new bounds?

Opt #5: Alpha-Beta Windows

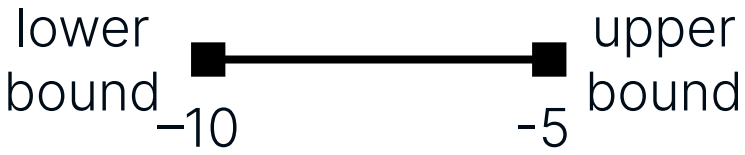


initial guess: α=-15 ⟶ alpha-beta result: -10

result > guess: failed high



Opt #5: Alpha-Beta Windows

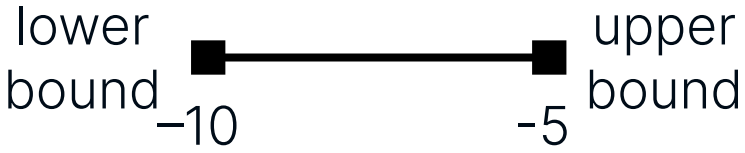


initial guess: α=-8 ⟶ alpha-beta result: -7

result > guess: failed high

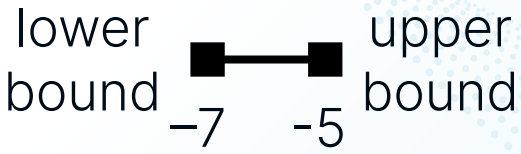
What's our new bounds?

Opt #5: Alpha-Beta Windows

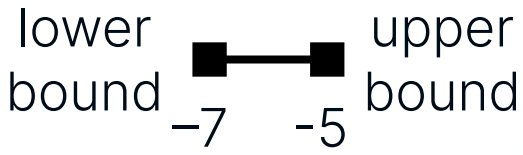


initial guess: α=-8 ⟶ alpha-beta result: -7

result > guess: failed high

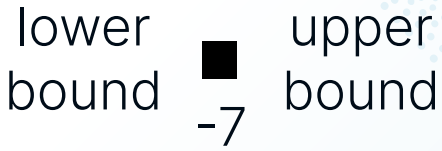


Opt #5: Alpha-Beta Windows



initial guess: α=-6, β=-5 ⟶ alpha-beta result: -7

result < guess: failed low



TRUE MINIMAX VALUE: -7

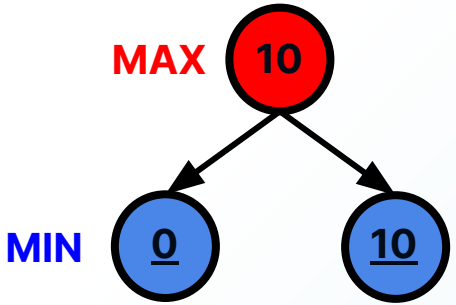
Opt #6: Iterative Deepening

Don’t immediately solve for minimax(depth=D); run minimax on the smaller depths first!

- find minimax(depth=1), then minimax(depth=2), ..., and finally minimax(depth=D)
- faster or slower? Depends on where the true value lies in the game tree
  - Sol’n: immediately stop after a certain amount of time, then return the best result of minimax()

Opt #6: Iterative Deepening

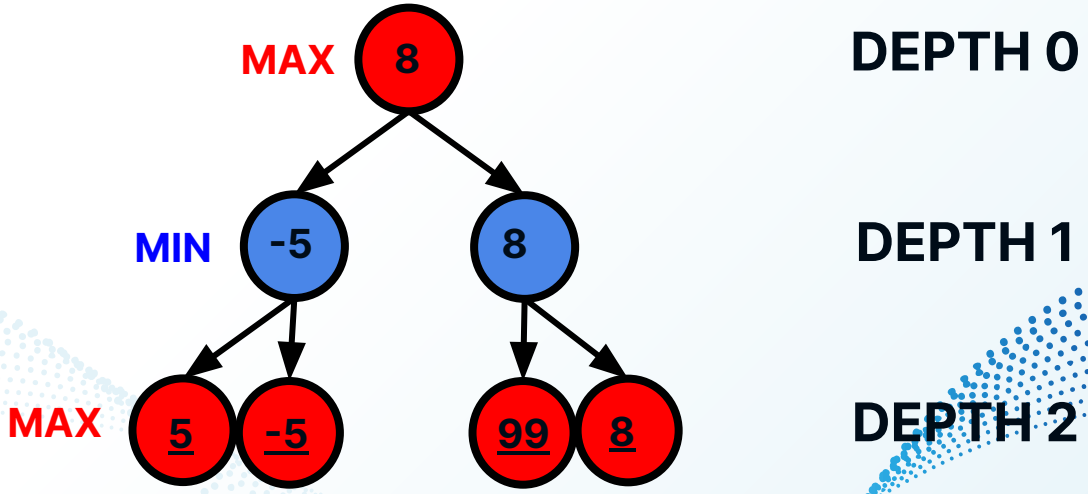
minimax(depth=1): 10



Opt #6: Iterative Deepening

minimax(depth=1): 10

minimax(depth=2): 8

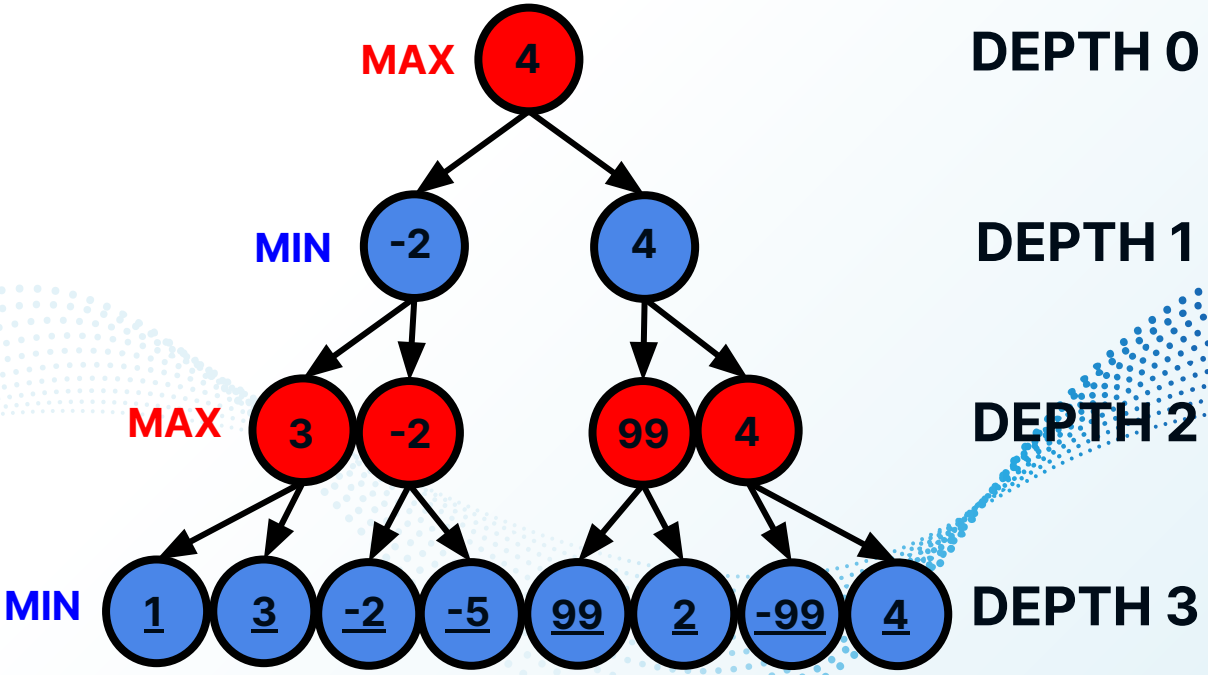


Opt #6: Iterative Deepening

minimax(depth=1): 10

minimax(depth=2): 8

minimax(depth=3): 4



Opt #6: Iterative Deepening

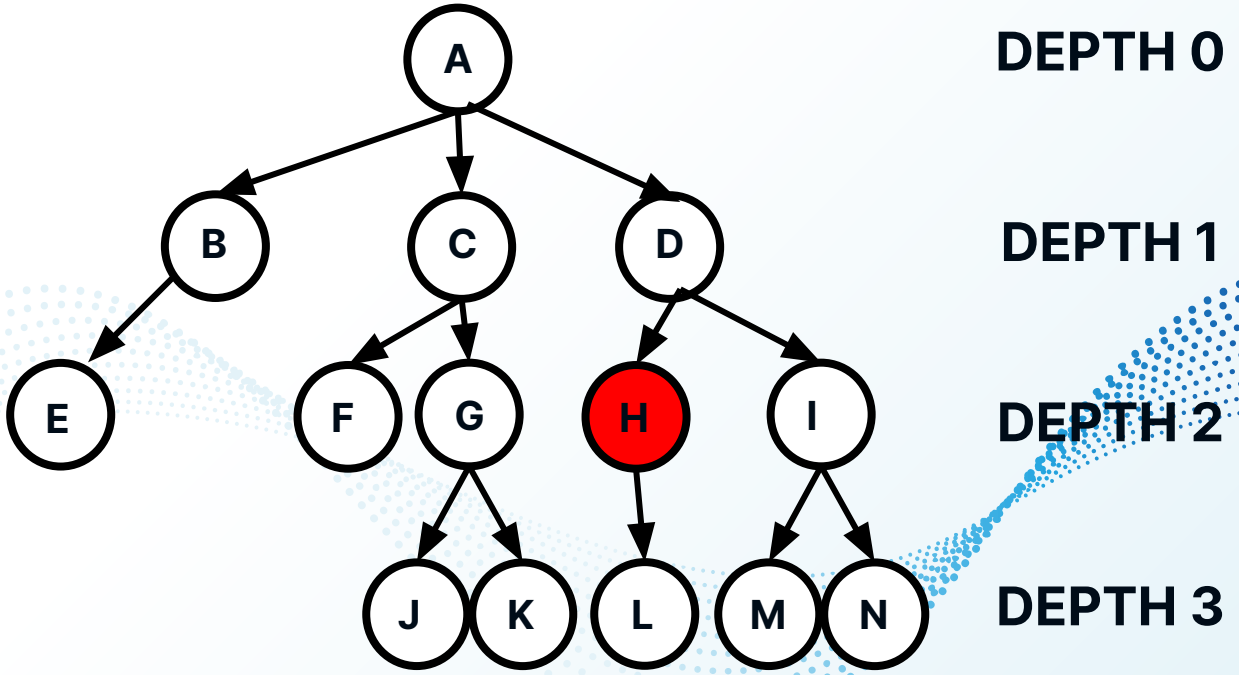
H = goal; stop here

What's the order of nodes explored if:

depth = 1?

depth = 2?

depth = 3?

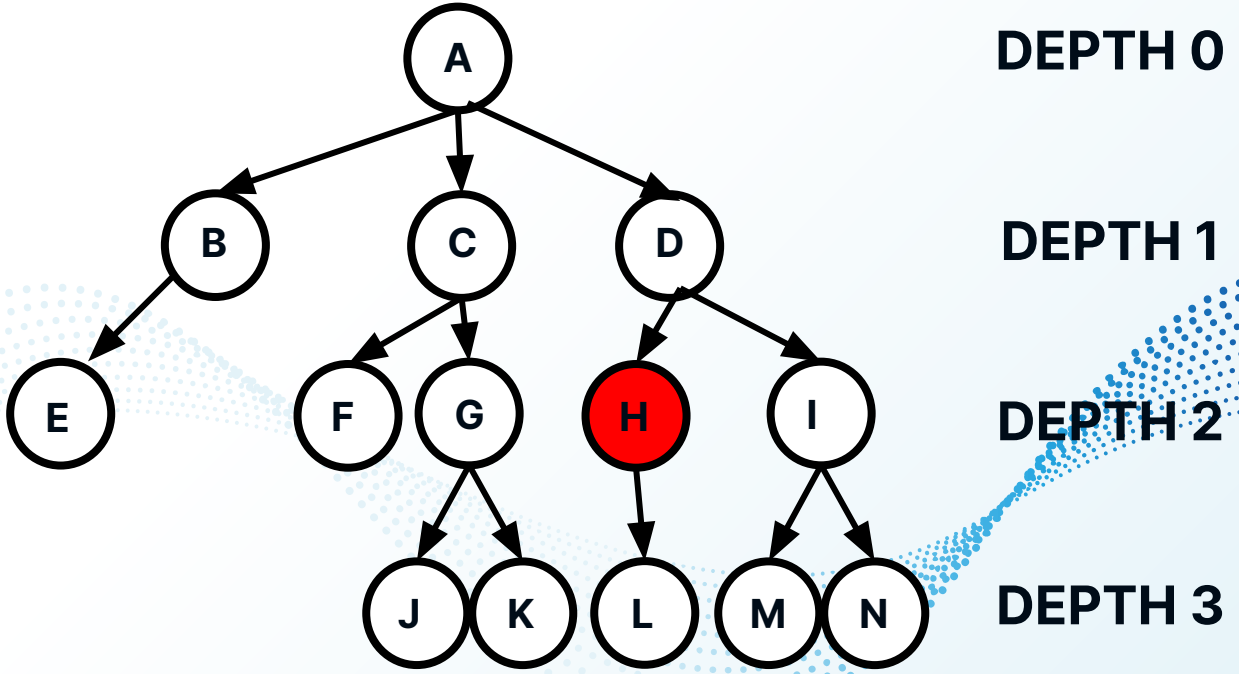


Opt #6: Iterative Deepening

depth = 1:
ABCD

depth = 2:
ABECFGDH

depth = 3:
ABECFGJKDH



Opt #6: Iterative Deepening

Advantages

- more efficient if sol’ns are at lower depths
- already have a good estimate from lower depths (which you can return if running out of time)
- combined w/ memoization, allows us to prune more b/c states at lower depths are stored

Combined Algorithm: MTD(f)

- Short for MTD(n,f) (Memory-enhanced Test Driver with node n and value f)
- **MTD(f) = minimax + alpha-beta pruning + null-window + iterative deepening + transposition table**
  - in addition, **add heuristics and better ordering**
- for games, MTD(f) is supposedly better than other algorithms such as PVS and NegaScout (full-size windows)
  - all better than minimax+pruning alone!

Combined Algorithm: MTD(f)

```
def MTDF(N, first_guess, depth):
    g = first_guess # anything, but more accurate = better
    upperBound = +∞ # anything ≥ max score
    lowerBound = −∞ # anything ≤ min score

    while lowerBound < upperBound: # iteratively narrow window
        β = max(g, lowerBound + 1)
        g = AlphaBetaWithMemory(N, β − 1, β, depth)
        # g tells us whether
        if g < β:  upperBound = g
        else:      lowerBound = g
    return g
```

Combined Algorithm: MTD(f)

```
def AlphaBetaWithMemory(N, α, β, depth):
   if in_TT(N): # N is in the transposition table
      if N.lowerBound >= β:  return N.lowerBound
      if N.upperBound <= α:  return N.upperBound
      α = max(α, N.lowerBound)
      β = min(β, N.upperBound)
   if depth==0:  g = evaluate(N) # using heuristics!
   elif N == MAXNODE:
      g, a = -∞, α
      child = firstchild(N)
      while g < β and child != None:
         g = min(g, AlphaBetaWithMemory(child, a, β, depth-1))
         a = max(a,g)
         child = nextbrother(child)
   elif N == MINNODE:
      g, b = +∞, β
      child = firstchild(N)
      while g > α and child != None:
         g = min(g, AlphaBetaWithMemory(child, α, b, depth-1))
         b = min(b,g)
         child = nextbrother(child)

   if g <= α:           N.upperBound = g
   if g >= β:           N.lowerBound = g
   if g > α and g < β:  N.upperBound = g, N.lowerBound = g # accurate minimax
   return g
```

Summary of Optimizations (so far)

1. OOP makes the code clearer and more organized
2. Heuristics will better estimate a state than just 0
3. Better move ordering lets us prune more worse moves
4. Transposition tables store and load states and their values that we came across earlier
5. Zero windows quickly find the upper and lower bounds of the true minimax value
6. Iterative deepening lets us explore shallow depths first before exploring further depths

More Optimizations

Even more connect-four-solving opts!

Opt #7: Bitboards

Instead of data structures like arrays, matrices, and lists, what if we could store everything in a single number?

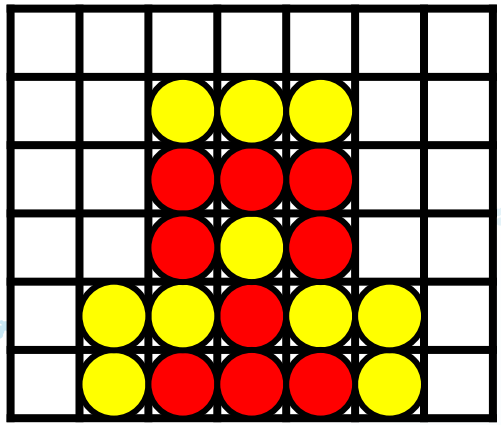
- Binary basics:
  - binary representation = base-2, where digits are 0 or 1 and represent sums of powers of 2
  - **Eg.** 20 (base 10) = 10100 (base 2)
    - 1(16) + 0(8) +1(4) + 0(2) + 0(1) = 20
  - **Eg.** 55 (base 10) = 110111 (base 2)
    - 1(32) + 1(16) + 0(8) + 1(4) + 1(2) + 1(1) = 55

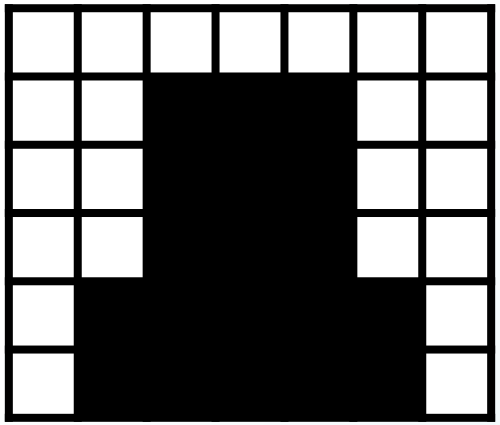
Opt #7: Bitboards

Binary Operators (all incredibly fast and optimized by the CPU)			
Operator Name	Operator Symbol	Definition	Example
AND	&	1 iff both bits are 1 0 otherwise	1101 & 1011 = 1001
OR	|	0 iff both bits are 0 1 otherwise	1001 & 1100 = 1101
XOR	^	1 iff both bits are different 0 otherwise	1101 ^ 1011 = 0110
LEFT SHIFT	<<	move all bits to the left by N	1101 << 3 = 1101000
RIGHT SHIFT	>>	move all bits to the right by N (and truncate any cutoffs)	100101 >> 2 = 1001

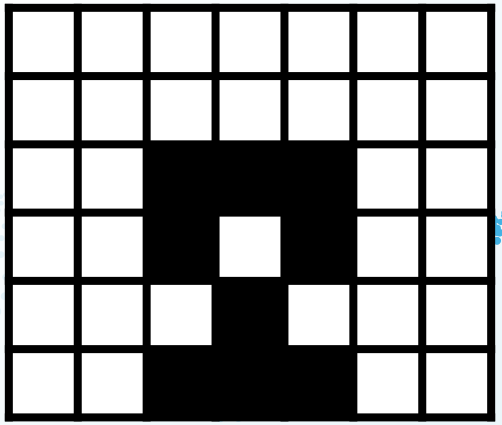
Opt #7: Bitboards

How to represent a board with just 0s and 1s:





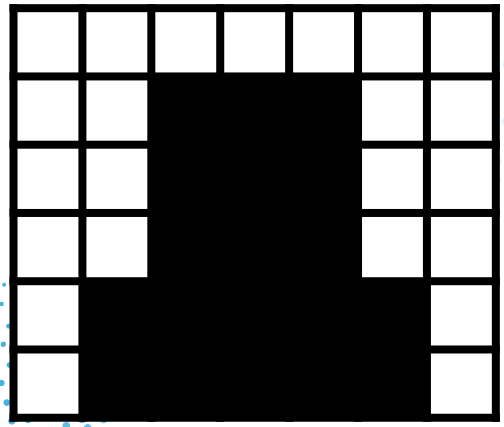
mask



position

Opt #7: Bitboards

6	13	20	27	34	41	48
5	12	19	26	33	40	47
4	11	18	25	32	39	46
3	10	17	24	31	38	45
2	9	16	23	30	37	44
1	8	15	22	29	36	43
0	7	14	21	28	35	42



set to 1 if column is full

0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	1	1	1	0	0
0	0	1	1	1	0	0
0	0	1	1	1	0	0
0	1	1	1	1	1	0
0	1	1	1	1	1	0

```
0000000110000011111001111100
1111100110000000000000
= 3332082970624
```

Opt #7: Bitboards

Every board function can now be performed very efficiently (in O(1)) using binary operators

```
def canPlay(col):
  top_mask = (1 << (HEIGHT - 1)) << col*(HEIGHT+1)
  return (mask & top_mask) == 0

def play(col):
  position ^= mask
  bottom_mask = 1 << col*(HEIGHT+1)
  mask |= mask + bottom_mask
  moves += 1
```

```
def gameIsOver(pos):
  m = pos & (pos >> (HEIGHT+1))  // horizontal
  if(m & (m >> (2*(HEIGHT+1)))): return true
  m = pos & (pos >> HEIGHT)  // diagonal 1
  if(m & (m >> (2*HEIGHT))): return true
  m = pos & (pos >> (HEIGHT+2))  // diagonal 2
  if(m & (m >> (2*(HEIGHT+2)))): return true
  m = pos & (pos >> 1)  // vertical
  if(m & (m >> 2)): return true

  return false
```

Opt #8: Better Move Ordering

Additional ways to pick better moves

- generally, moves at/near the middle are better
- avoid losing moves - find your opponent's winning moves and always block them
- score every move (using heuristics), then sort them efficiently

Opt #9: Better Transposition Tables

Additional ways to make a better transposition table

- have a very large table (eg. 1 million buckets)
  - tradeoff: uses up a lot of memory space
- make a good hash function that reduces the number of collisions (best is almost perfectly random)
  - less collisions = quicker average lookups
- make the table size a large prime number
  - less collisions and requires less memory space
- store both upper and lower bounds
  - though not much improvement, and more complex

Solving Connect Four

The optimal optimized strategy!

Connect Four Optimizations

Download connectfour.py and connectfour_gilles from GitHub

- Scan through the files to understand the structure
  - I don't blame you if you don't understand it!
- Try playing it for yourself by running the code
- Learn more about Connect Four optimizations at:
  - http://blog.gamesolver.org/
  - https://towardsdatascience.com/creating-the-perfect-connect-four-ai-bot-c165115557b0

Solving ???

Optimize our previous games!

Game Optimizations

Download tictactoe_solution.py or sim_solution.py from GitHub

- Try to fully understand how that file works
- Add 1+ optimization to the game and/or AI algorithm
  - Eg. bitboards, iterative deepening, heuristics, move-ordering, transposition table, MTD(f)
  - Each optimization might require adding 1 line, or changing the entire file!
- This is an individual project; we want to see unique improvements to the game!

Testing Your Code's Improvement

```
import time

...

def Player1Strategy(params):
    start_time = time.time()
    bestMove = YourOptimizedAIStrat(params)
    end_time = time.time()
    print("Time: %f" % (end_time - start_time))
```

Report the differences in runtime between the _solution.py algorithm and your optimized algorithm!

To-do

Things to do by the next workshop:

- Glance over the connectfour files on GitHub
- Read more about Connect Four optimizations
  - see slide 67
- Complete your optimized game file
  - We'll be doing demonstrations next week

Thanks for coming!