Back-Tracking

Thomas Schwarz, SJ

Complete Enumeration

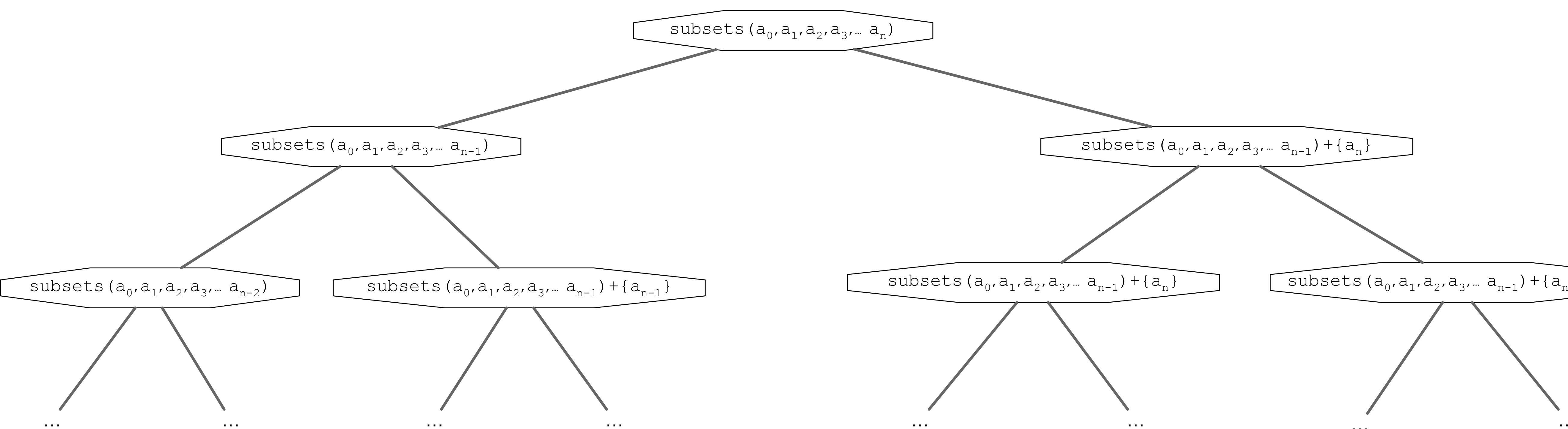
- You are given:
    - A set of numbers, e.g. $\mathbb{S} = \{1,5,12,14,19,20,21\}$
    - A target number t
- Your task is to find a subset of $\mathbb{S}$ such that the sum of the numbers in the subset is as close to t as possible.

Complete Enumeration

- Complete enumeration solves this by
  - creating all subsets
  - selecting the one that works best
- One possibility is to use recursion for complete enumeration

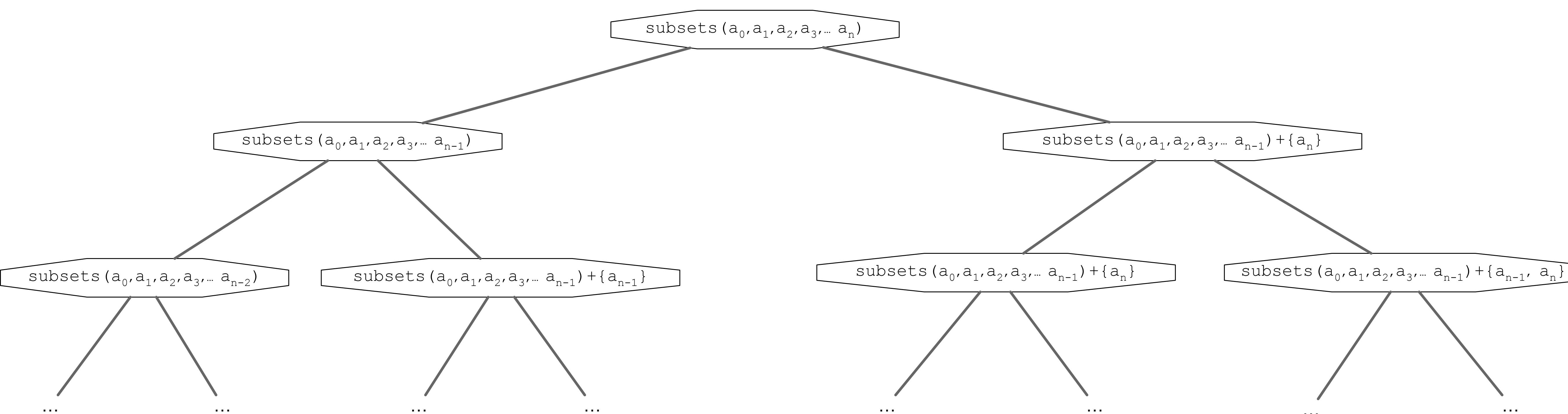
Complete Enumeration

- Base case:
  - Subsets of the empty set are just the empty set
  - Subsets of a set with one element x are just $\emptyset$, $\{x\ ]$



Complete Enumeration

- Recursive Case:
  - Subsets of the set $\{a_1, \ldots, a_n\}$ are:
    - Subsets of $\{a_1, \ldots, a_{n-1}\}$
    - Subsets consisting of a subset of $\{a_1, \ldots, a_{n-1}\}$ and a_n



Complete Enumeration

- How to represent sets?
  - Python has a type sets, but the elements need to be hashable
  - And sets are not hashable
  - Could use frozen_sets, but these are ugly
- So, create the set of subsets as a list

Complete Enumeration

- Implementation:

```
def subsets(a_list):
    if len(a_list) == 0:
        return []
    if len(a_list) == 1:
        return [[], [a_list[-1]]]
    lst = a_list[-1]
    menge = subsets(a_list[:-1])
    return menge + [ x+[lst] for x in menge]
```

Complete Enumeration

- Example: $\mathbb{S} = \{1,5,12,14,19,20,21\}$ target 37:

```
lista = [1, 5, 12, 14, 19, 20, 21]

for subset in subsets(lista):
    if sum(subset) == 37:
        print(subset)
```

- ```
  [1, 5, 12, 19]
  [5, 12, 20]
  ```
  ```

Complete Enumeration

- If you want to find the best approximation, you need to remember the best value so far

```
def find(lista, target):
    best = sum(lista)+1
    best_seen = []
    for subset in subsets(lista):
        if abs(sum(subset) - target) < best:
            best = abs(sum(subset) - target)
            best_seen = subset
    return best, best_seen
```

Complete Enumeration

- Example: Target is 43
- Best: 1, [5, 19, 20]

Complete Enumeration

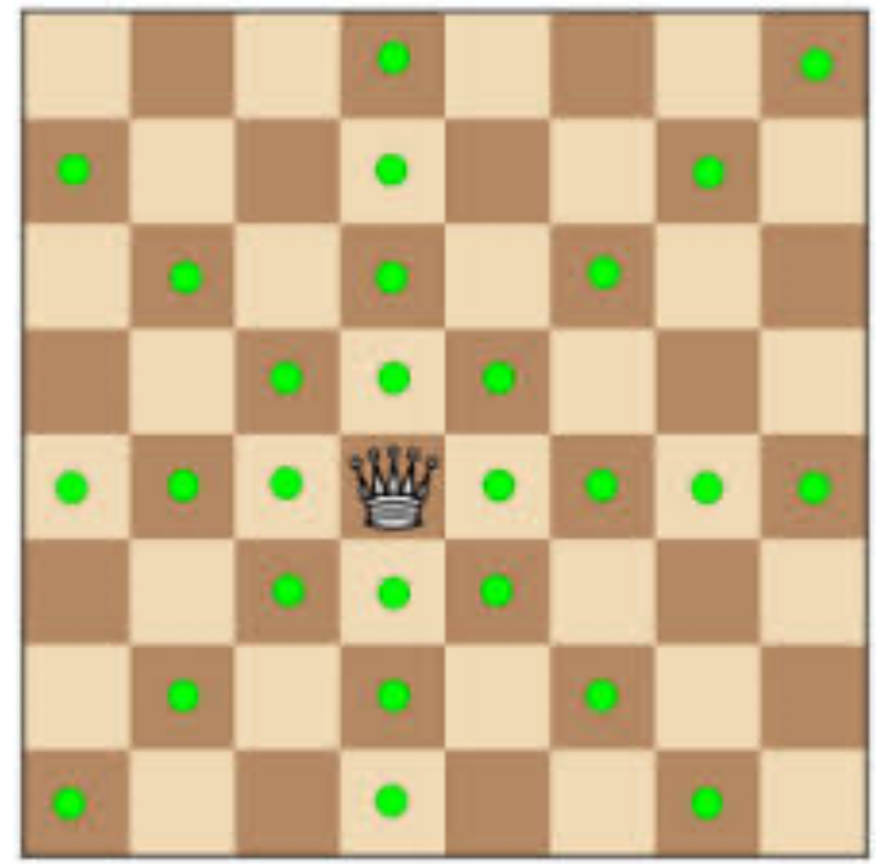
- Complete enumeration of subsets generates 2^n subsets
    - Therefore, is exponential
- In general: complete enumeration with recursion creates a call tree with b^n or b^{n+1} leaves

Back Tracking

- Idea:
  - We do not always need to go down to the leaves of the tree, but can stop earlier

Back Tracking

- Example:
  - The *n*-queens problem
    - Place *n*-queens on a $n \times n$ chessboard so that no queen threatens any other
    - Queens can move vertically, horizontally, and diagonally



Back Tracking

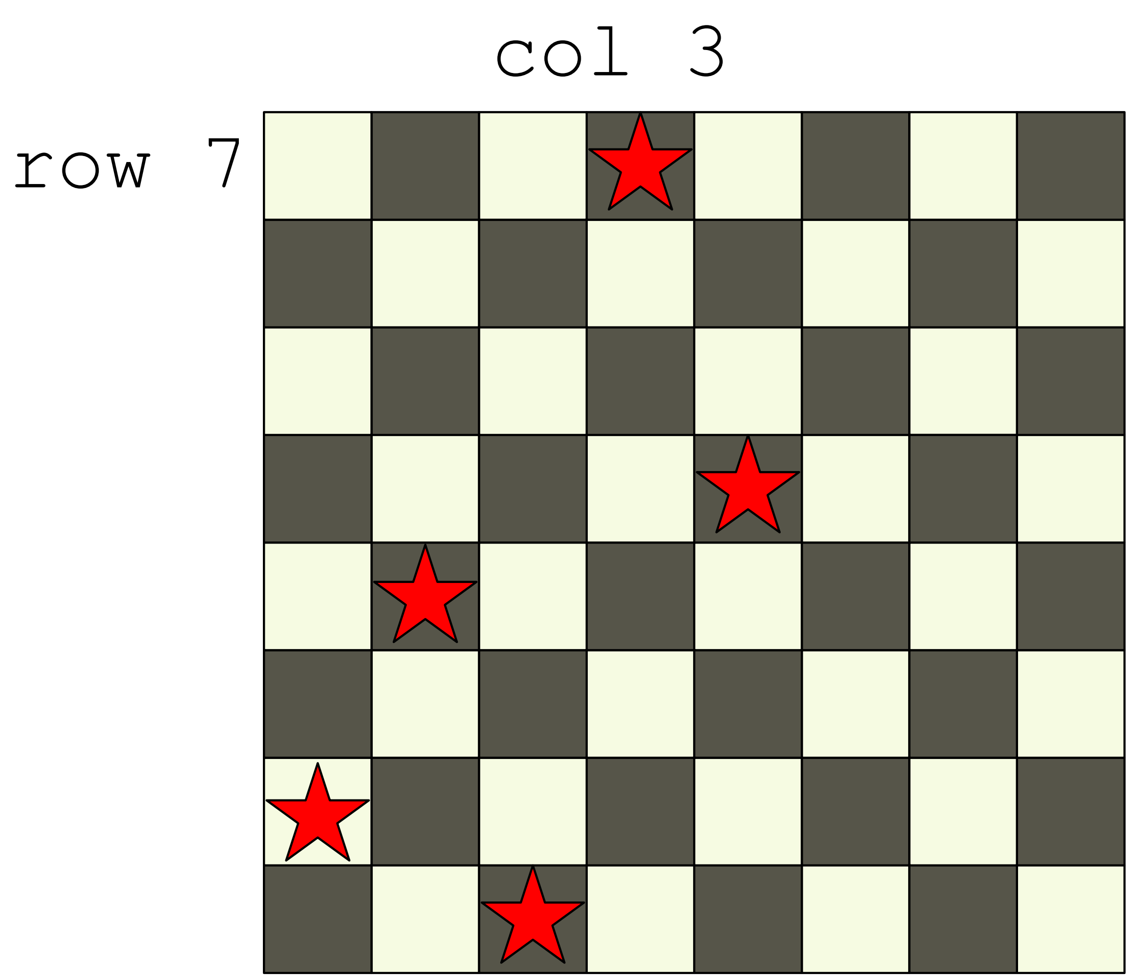
- Strategy:
    - We notice that there can be only one queen per column
    - And that there has to be one in every column to get the total number to n

Back Tracking

- Add queen to a partial solution
  - Check whether queen placement is possible
    - If not, stop this branch in the tree
- Trick is to use recursion so that we do not have to administer walking up and down the tree

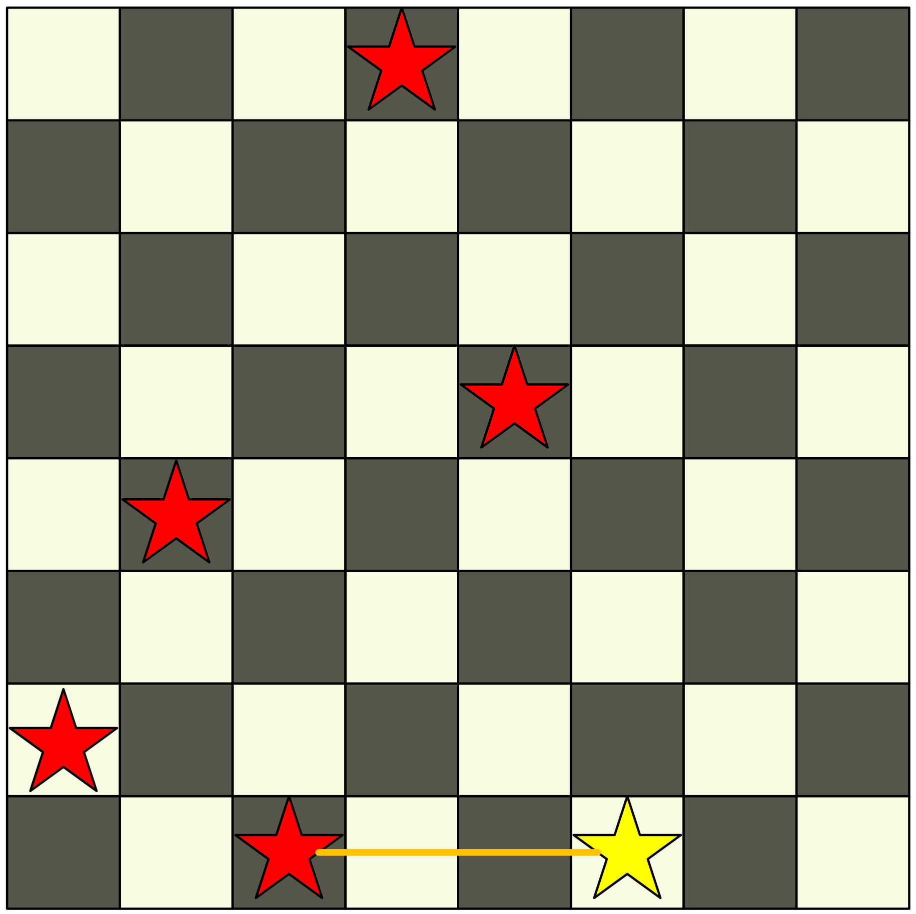
Back Tracking

- We encode the problem by having a list `board`
- i th queen is located in column i and row `board[i]`
  - E.g. `board = [1,3,0,7,4]`



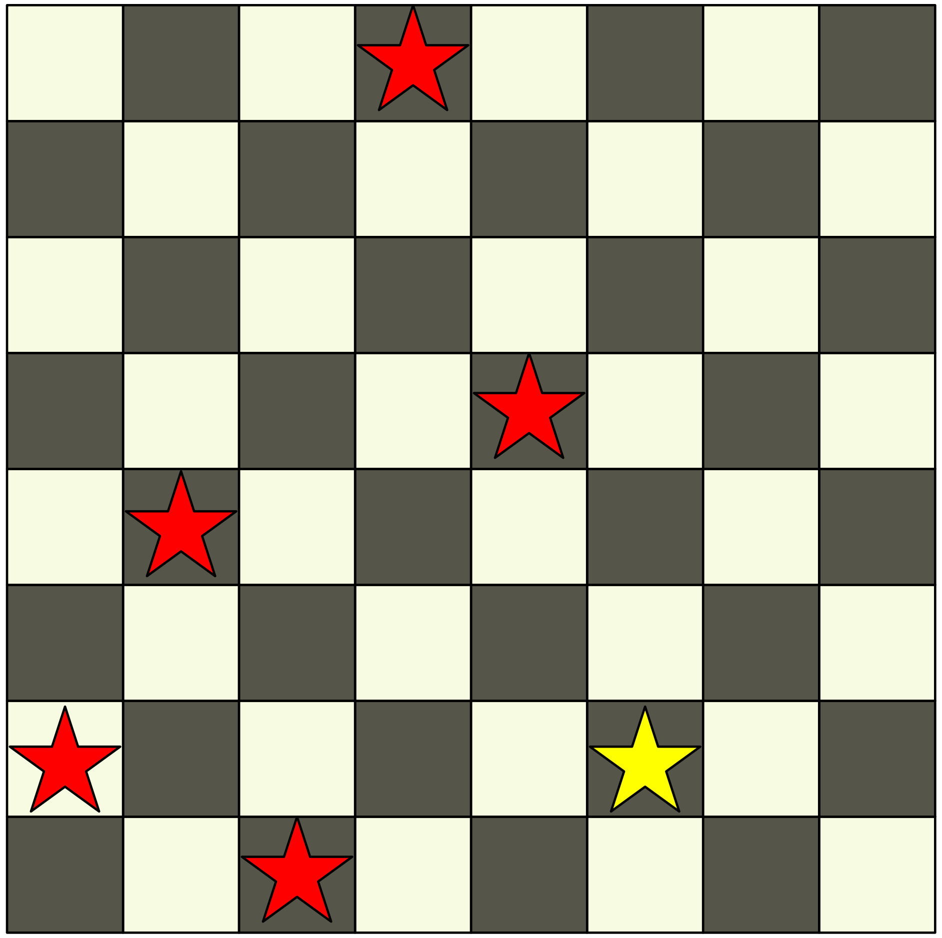
Back Tracking

- E.g. `board=[1,3,0,7,4]`
  - We then assign the next queen in column 5
    - We try out: 0, 1, 2, … , 7
      - 0 does not work



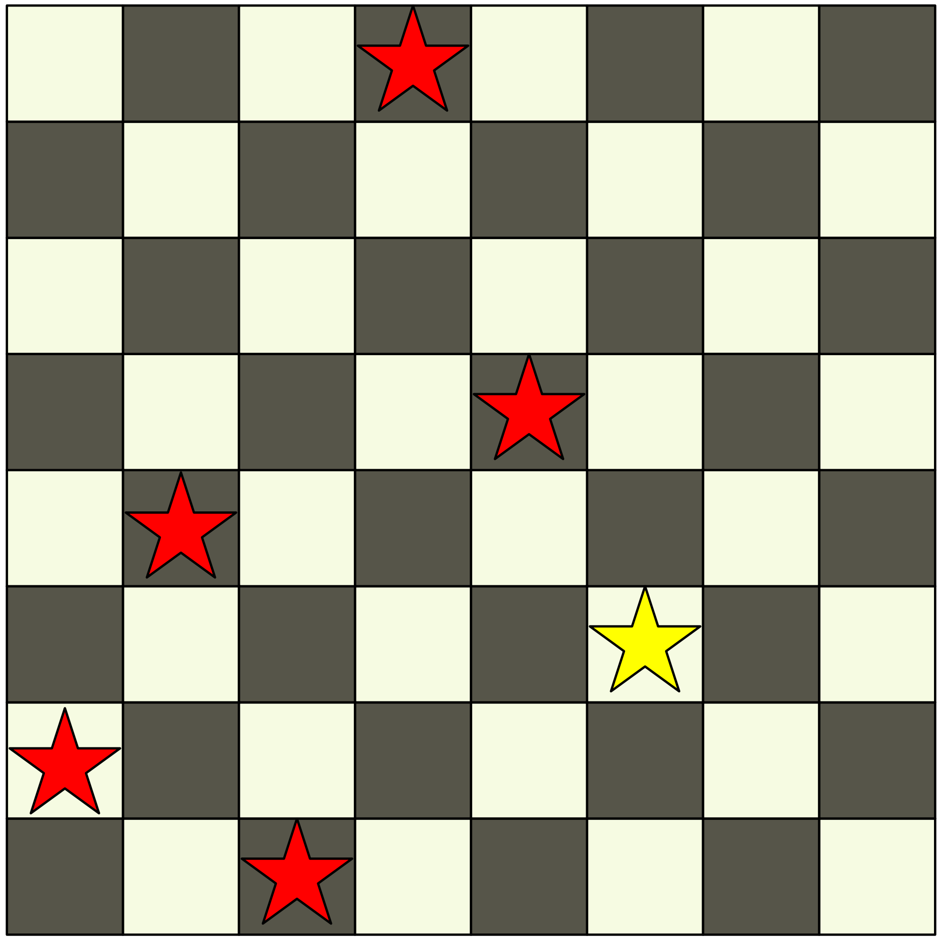
Back Tracking

- E.g. `board=[1,3,0,7,4]`
  - We then assign the next queen in row 5
    - We try out: 0, 1, 2, … , 7
      - 1 does not work



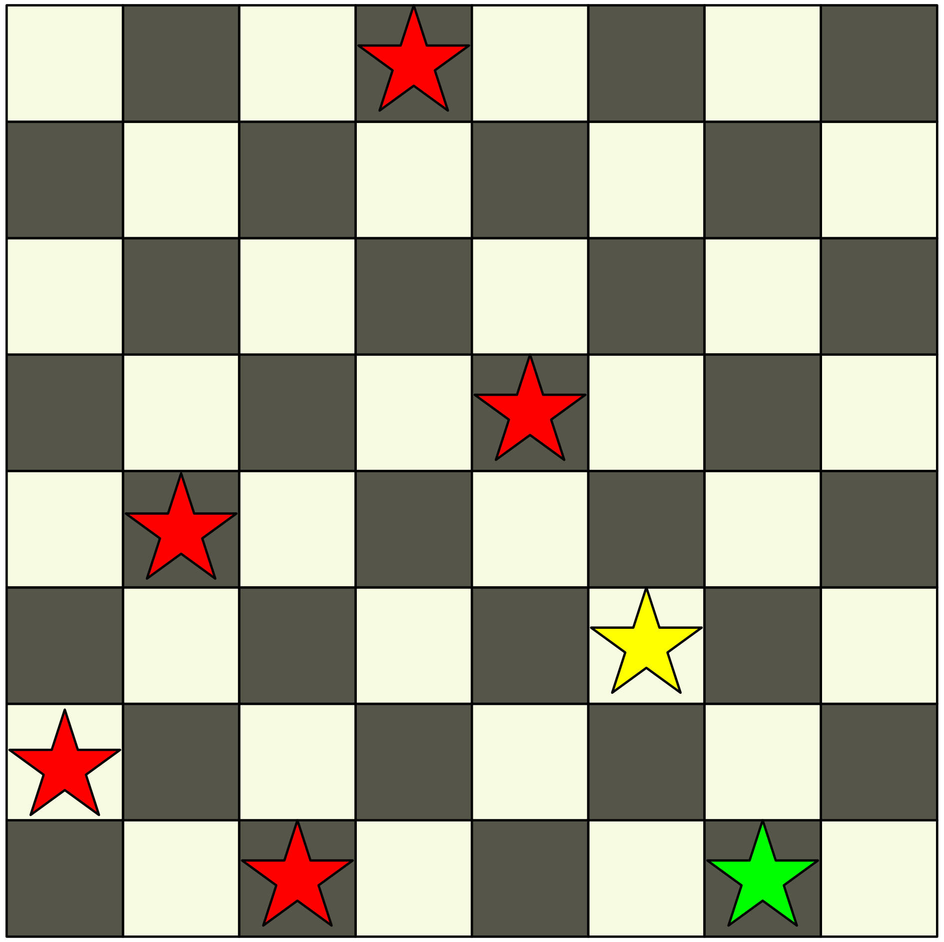
Back Tracking

- E.g. `board=[1,3,0,7,4]`
  - We then assign the next queen in row 5
    - We try out: 0, 1, 2, … , 7
    - 2 does work
    - `board=[1,3,0,7,4, 2]`



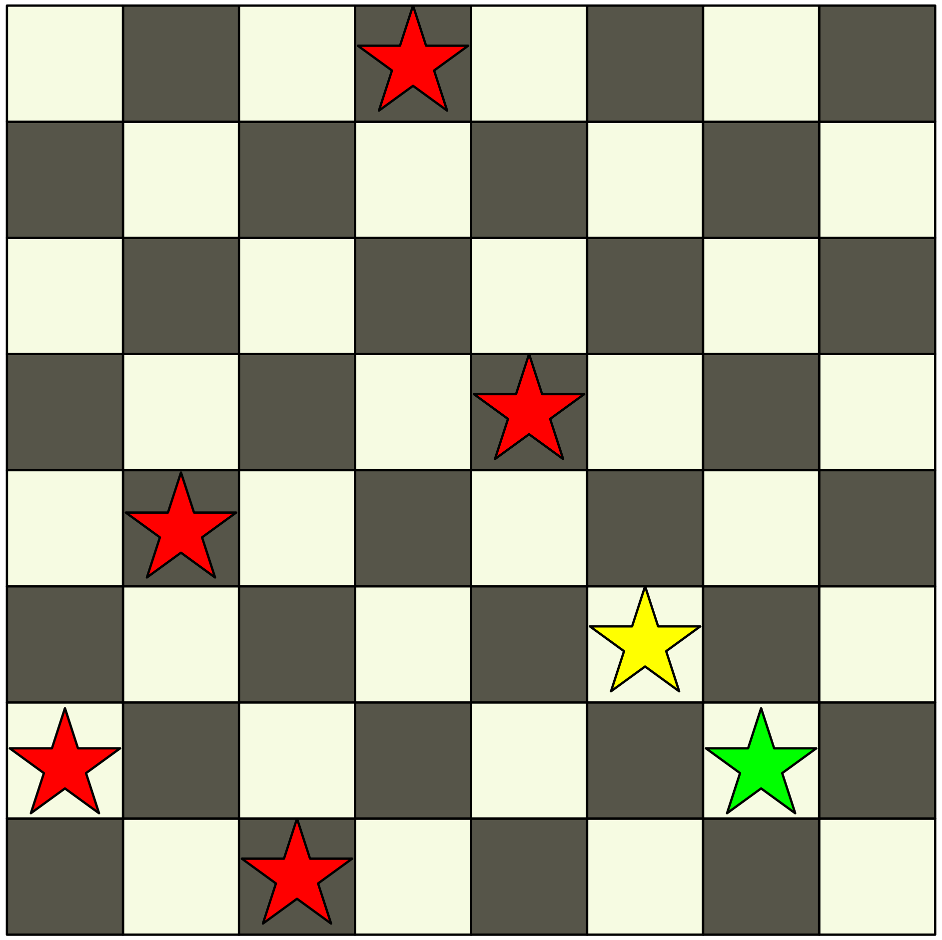
Back Tracking

- E.g. `board=[1,3,0,7,4,2]`
  - We then assign the next queen in column 6
    - We try out: 0
    - 0 does not work



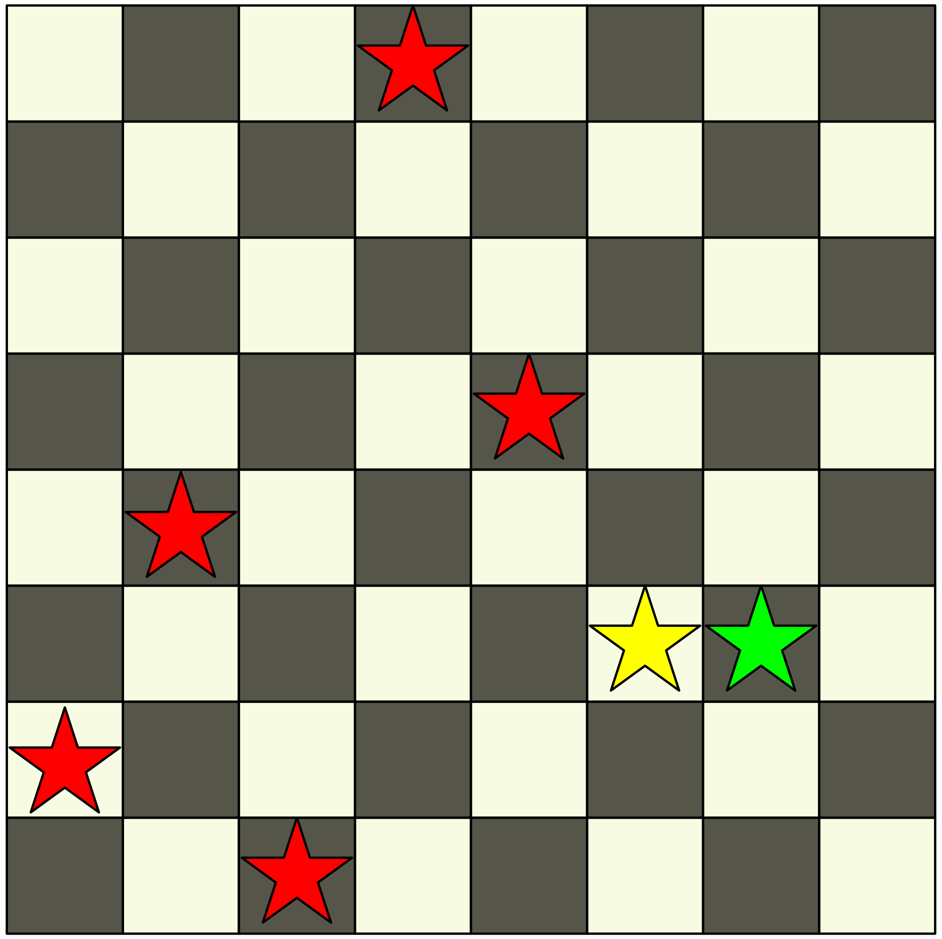
Back Tracking

- E.g. `board=[1,3,0,7,4,2]`
  - We then assign the next queen in column 6
    - We try out: 1
    - 1 does not work



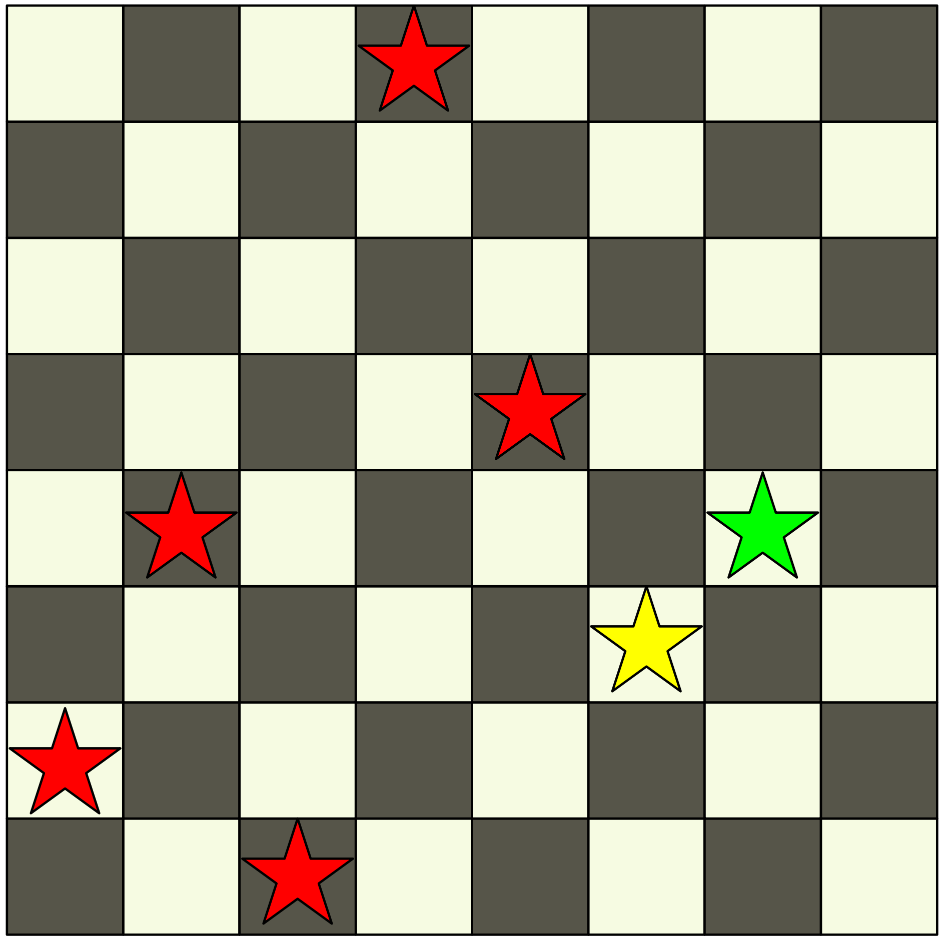
Back Tracking

- E.g. `board=[1,3,0,7,4,2]`
  - We then assign the next queen in column 6
    - We try out: 2
    - 2 does not work



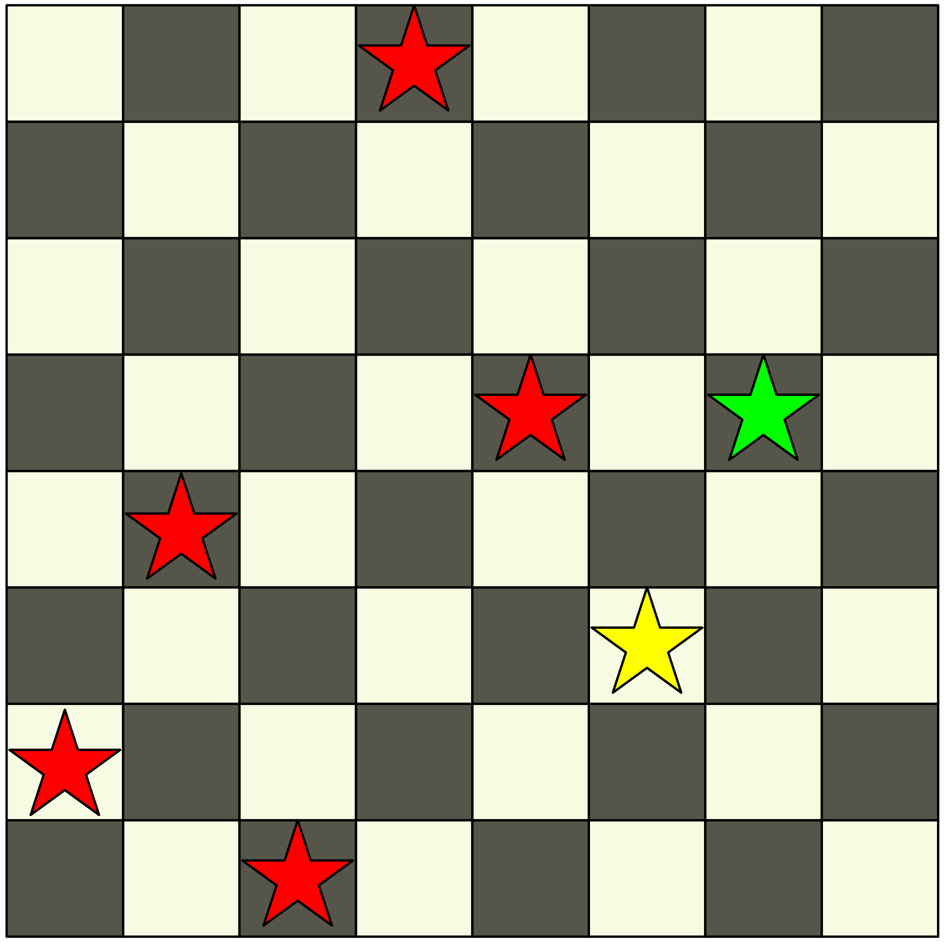
Back Tracking

- E.g. `board=[1,3,0,7,4,2]`
  - We then assign the next queen in column 6
    - We try out: 3
    - 3 does not work



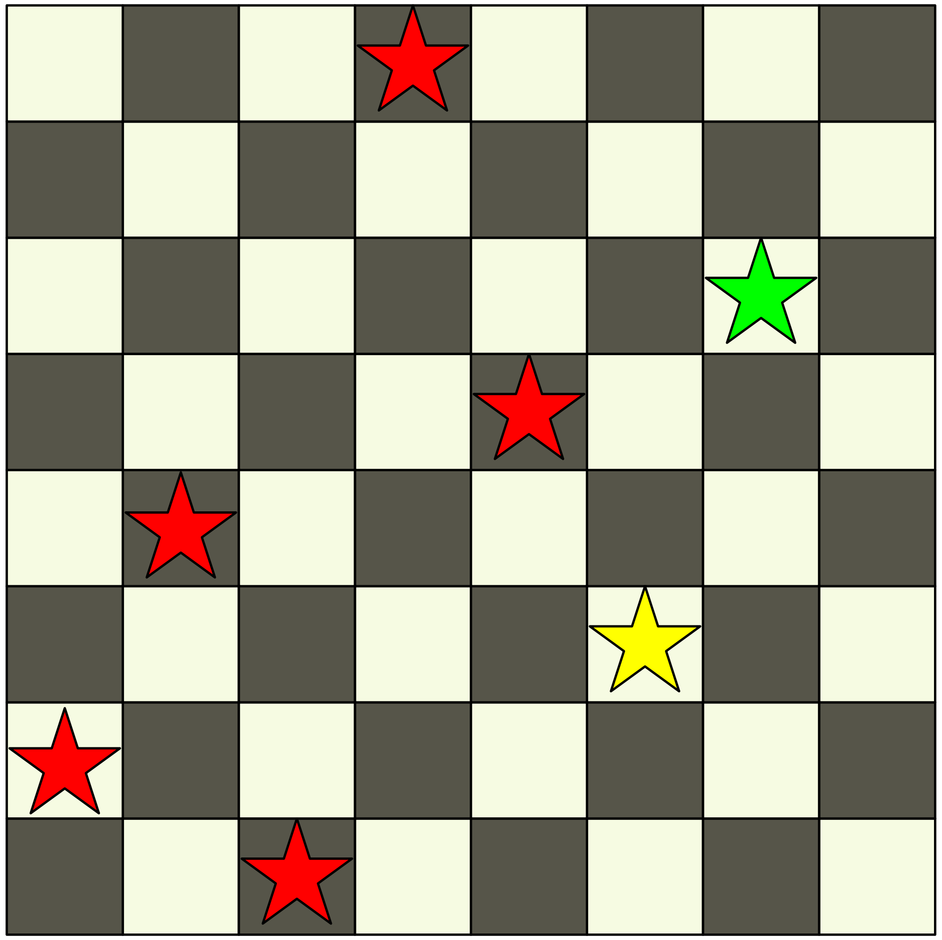
Back Tracking

- E.g. `board=[1,3,0,7,4,2]`
  - We then assign the next queen in column 6
    - We try out: 4
    - 4 does not work



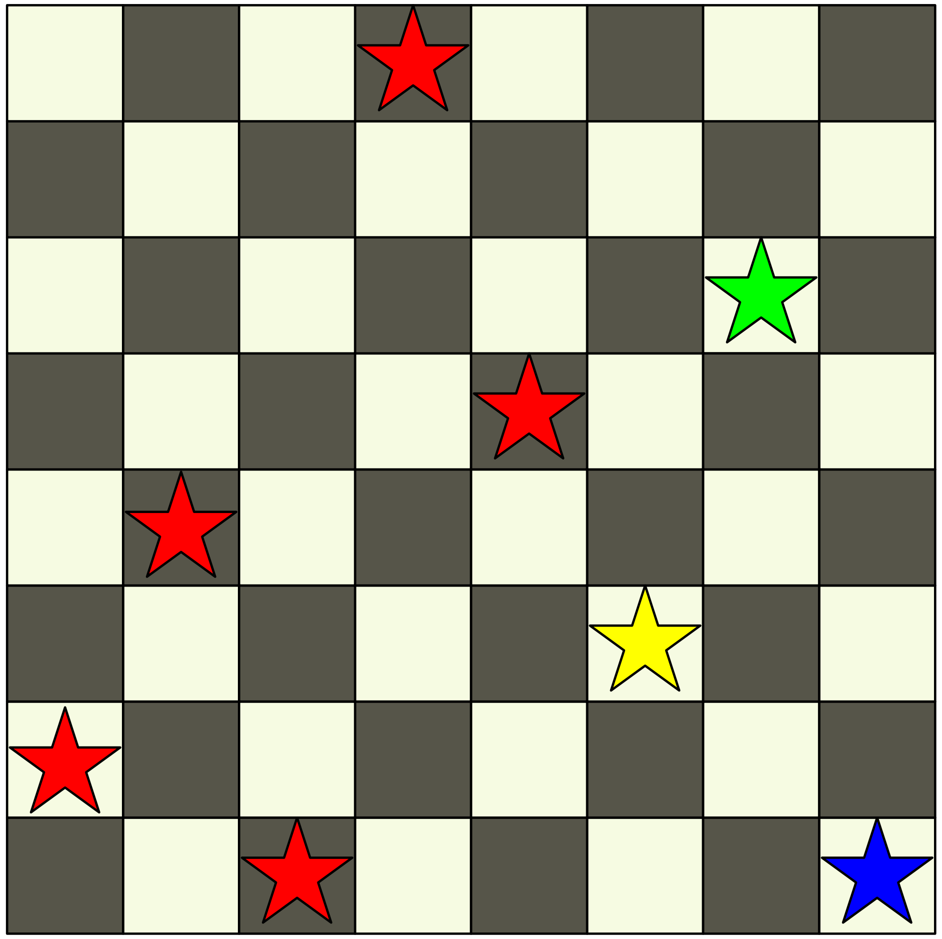
Back Tracking

- E.g. `board=[1,3,0,7,4,2]`
  - We then assign the next queen in column 6
    - We try out: 5
    - 5 does work
    - `board=[1,3,0,7,4,2,5]`



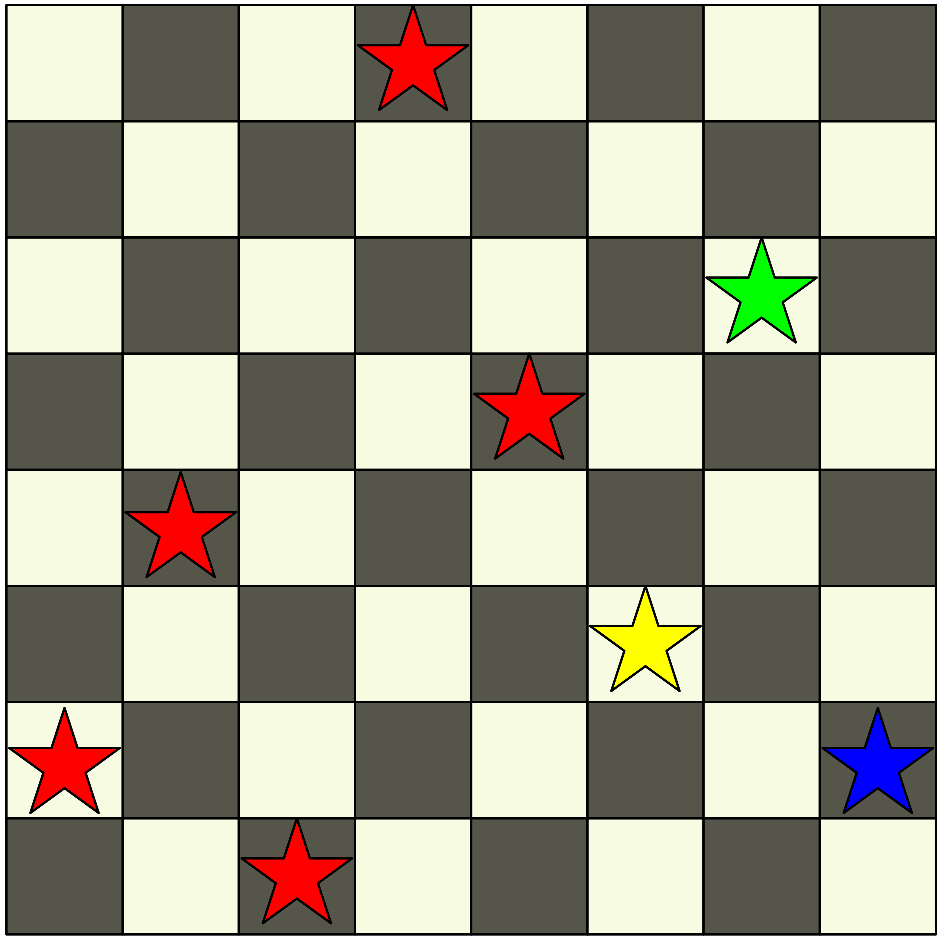
Back Tracking

- E.g. `board=[1,3,0,7,4,2,5]`
  - We then assign the next queen in column 7
    - We try out: 0
    - 0 does not work



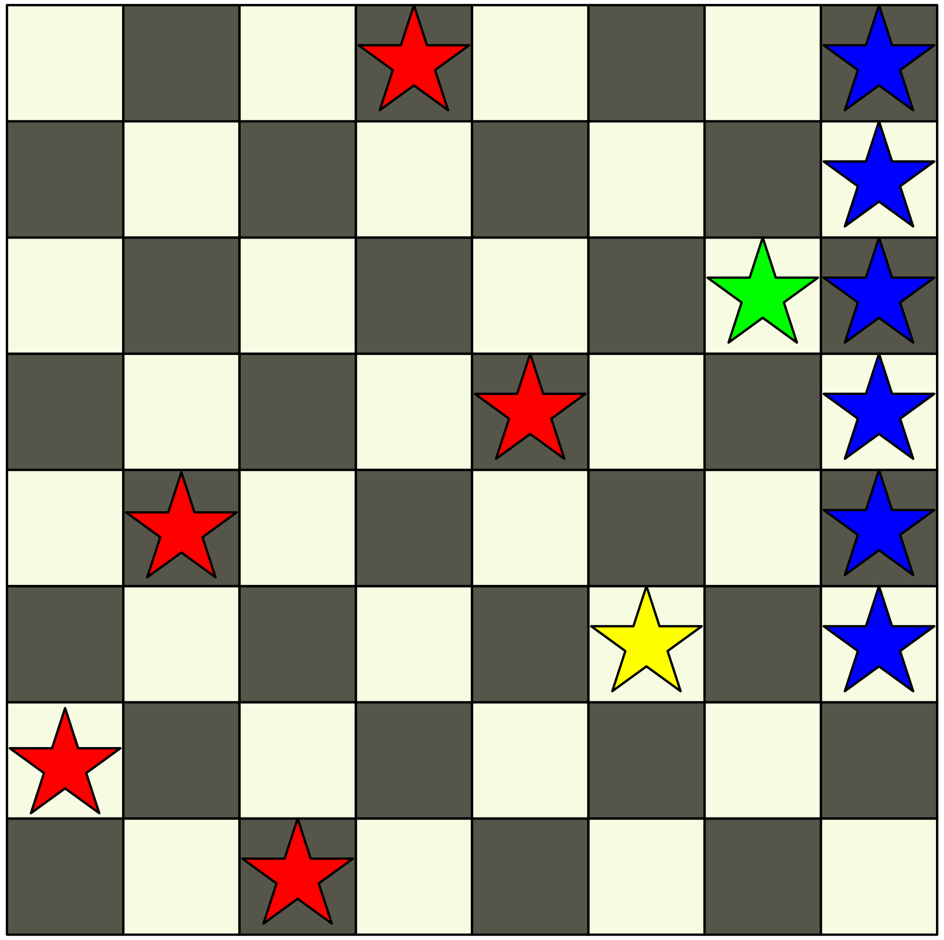
Back Tracking

- E.g. `board=[1,3,0,7,4,2,5]`
  - We then assign the next queen in column 7
    - We try out: 1
    - 1 does not work



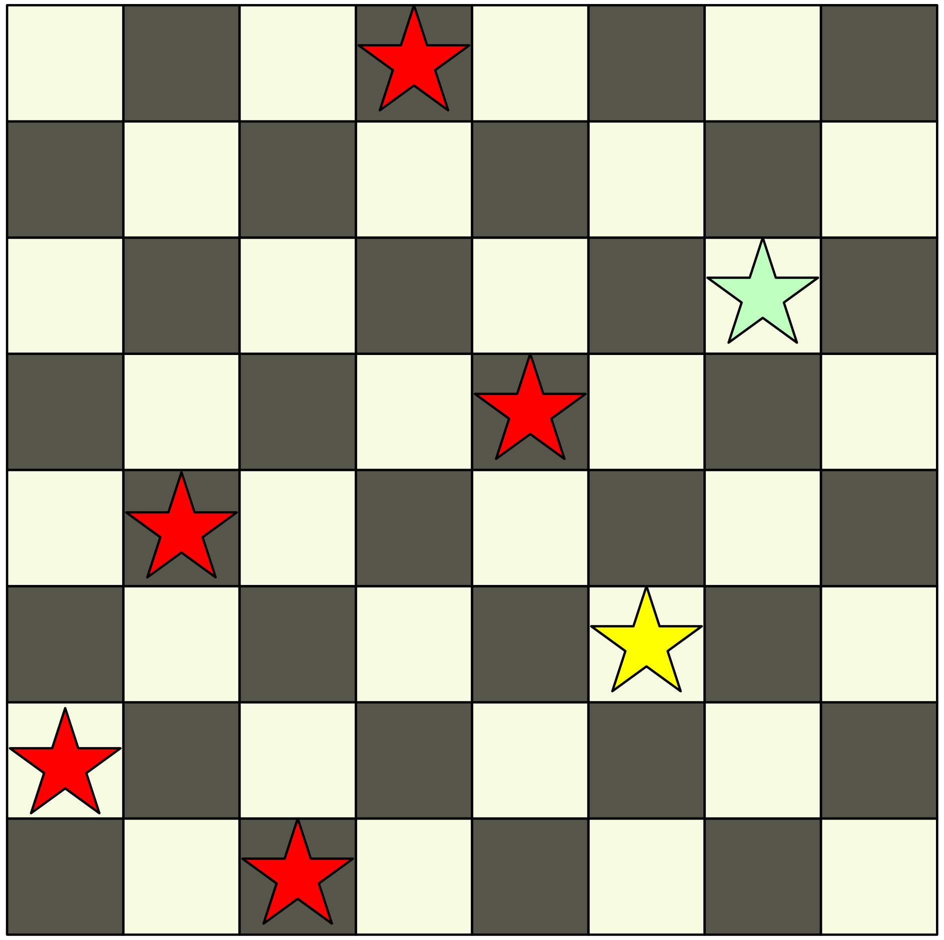
Back Tracking

- E.g. `board=[1,3,0,7,4,2,5]`
  - We then assign the next queen in column 7
    - We try out: 2, 3, ..., 7
    - none works



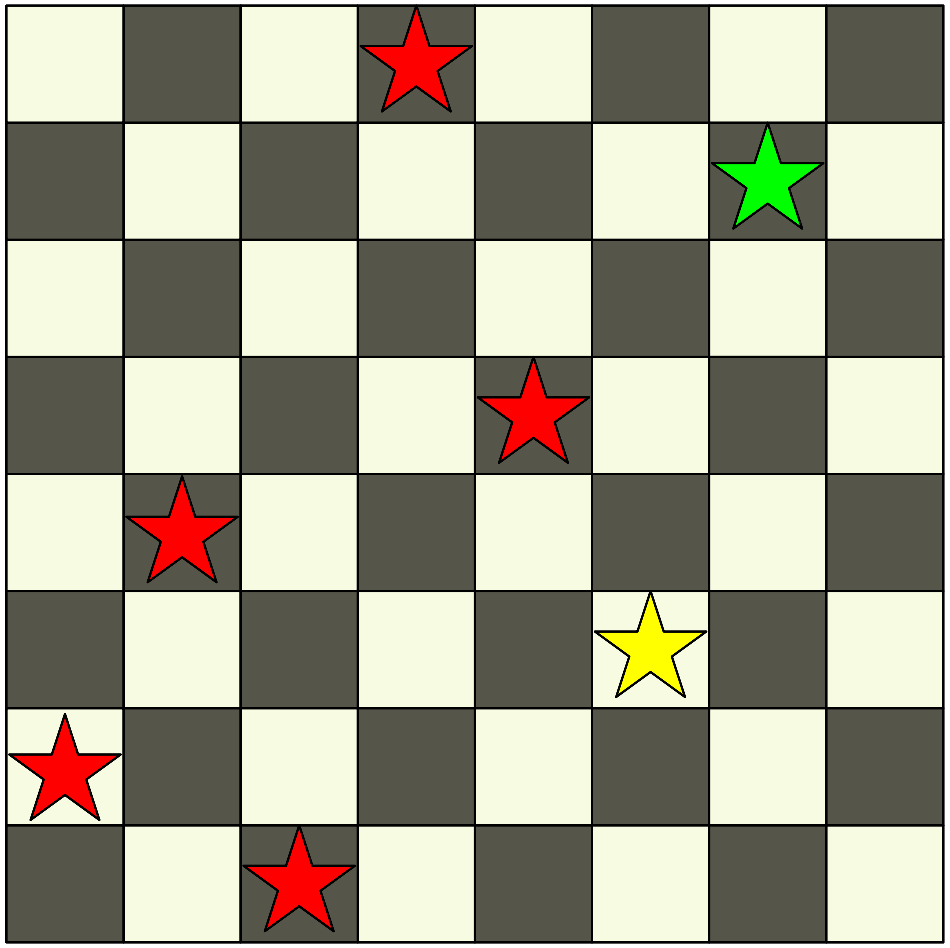
Back Tracking

- E.g. `board=[1,3,0,7,4,2,5]`
  - We now remove 5
  - `board=[1,3,0,7,4,2]`



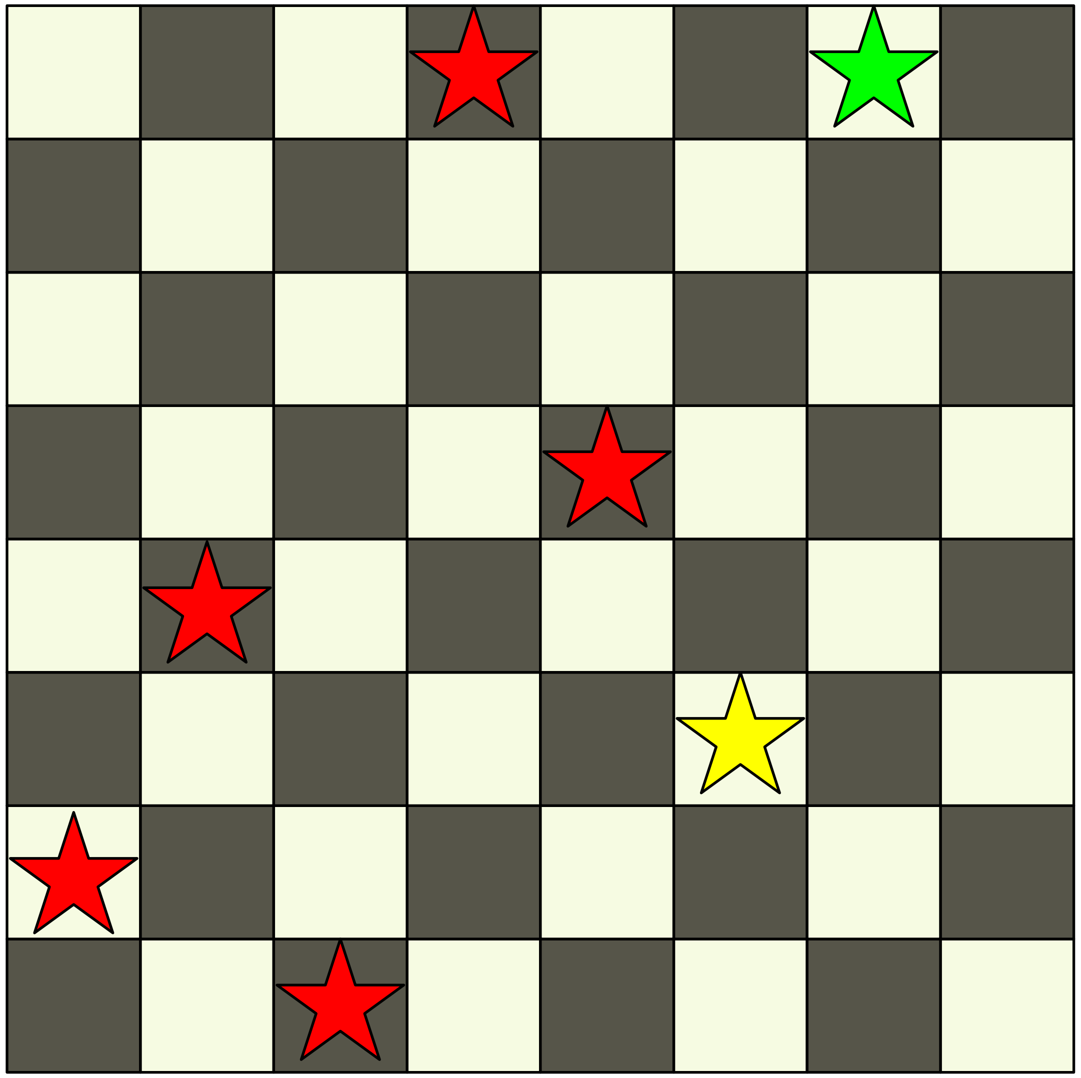
Back Tracking

- E.g. `board=[1,3,0,7,4,2,5]`
  - We now remove 5
  - `board=[1,3,0,7,4,2]`
  - And go to the next one
  - `board=[1,3,0,7,4,2,6]`
  - which does not work



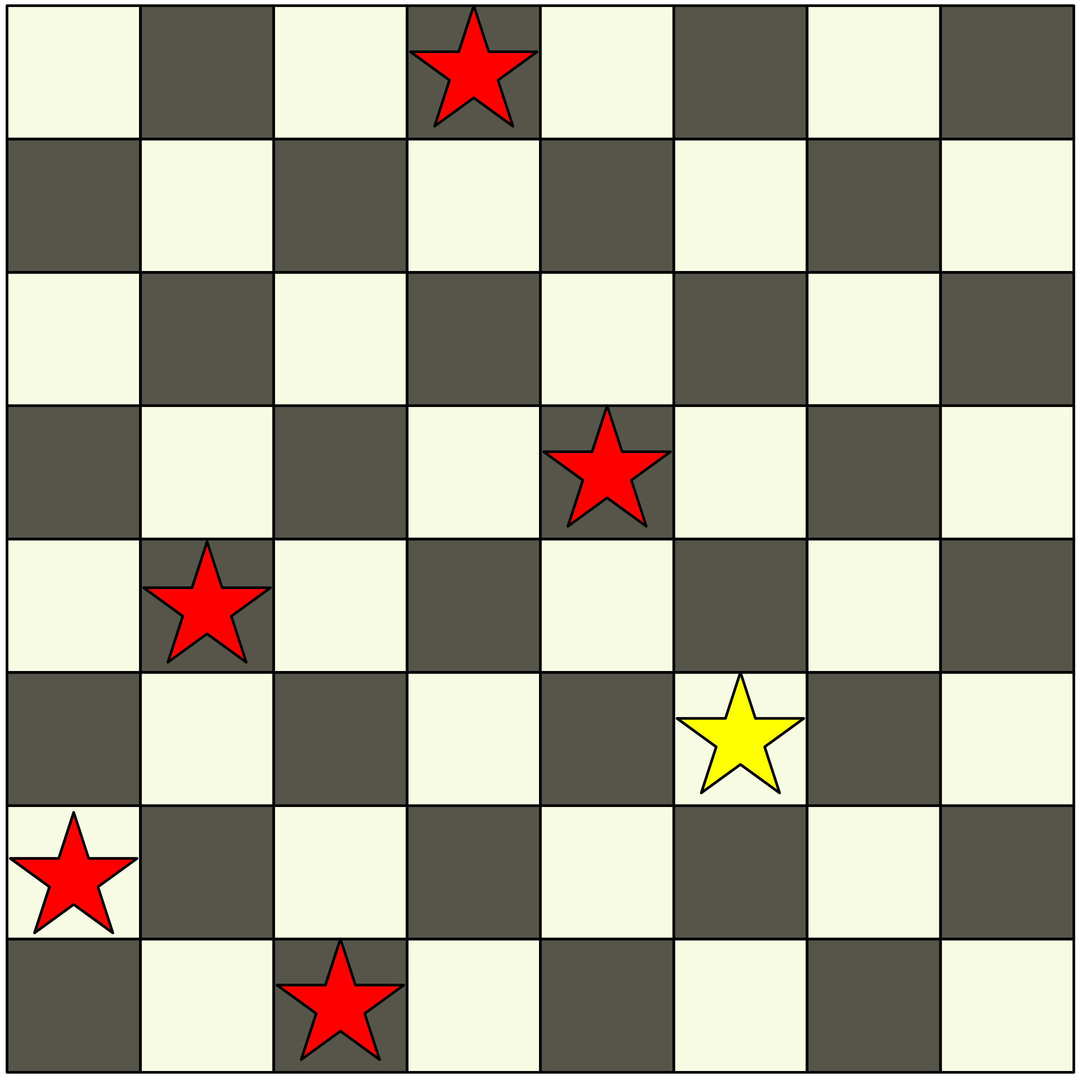
Back Tracking

- E.g. `board=[1,3,0,7,4,2,5]`
  - We now remove 5
  - `board=[1,3,0,7,4,2]`
  - And go to the next one
  - `board=[1,3,0,7,4,2,6]`
  - which does not work
  - so we try the next one
  - `board=[1,3,0,7,4,2,7]`
  - which does not work



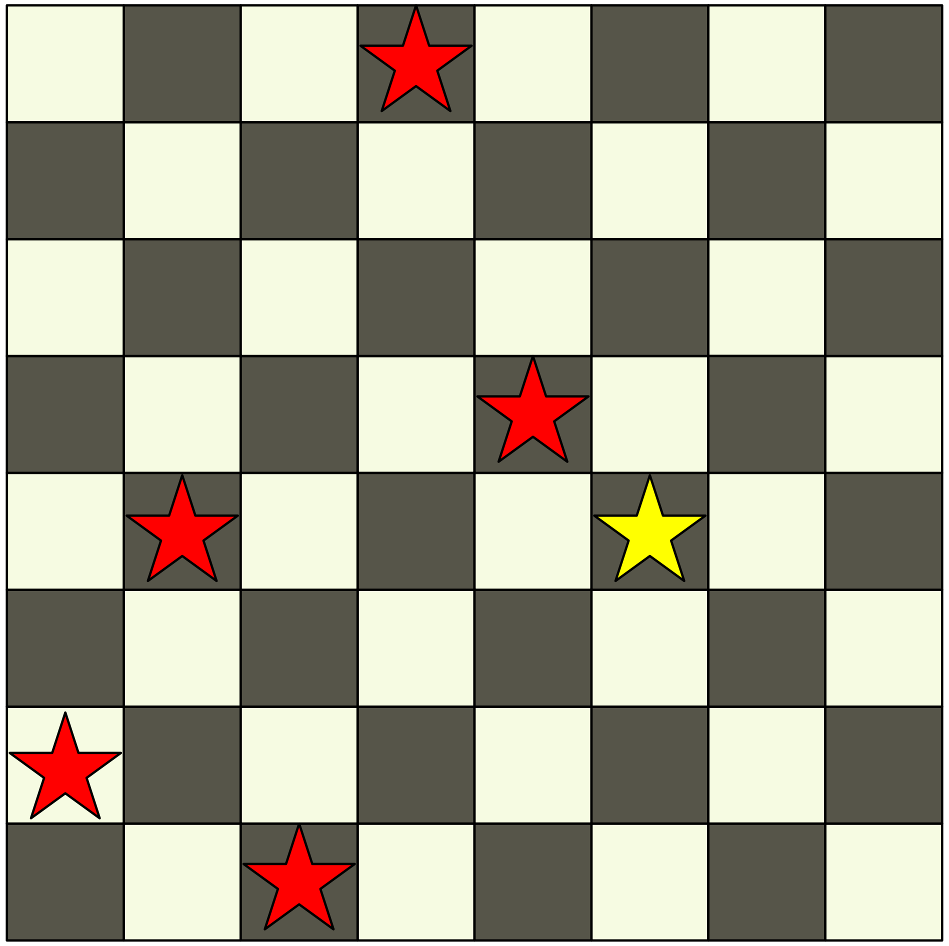
Back Tracking

- E.g.
  `board=[1,3,0,7,4,2,?]`
  - All possibilities are exhausted
  - We return and try the next position for column 5



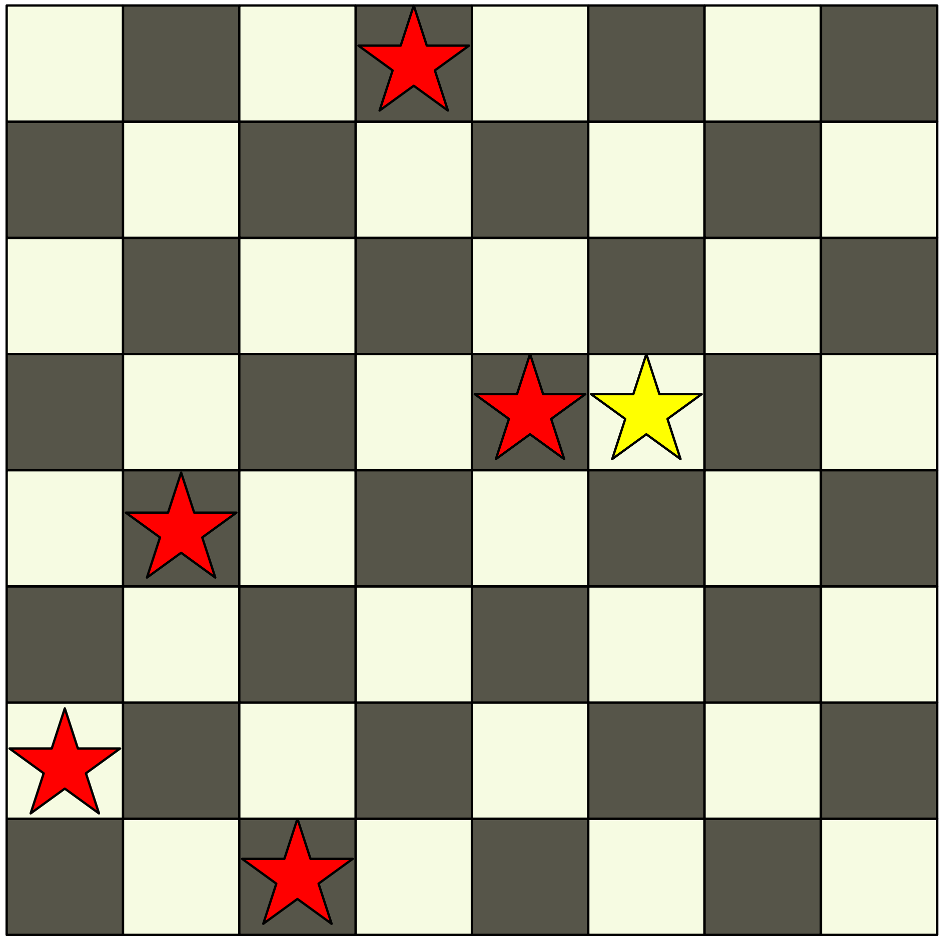
Back Tracking

- E.g.
  `board=[1,3,0,7,4,3]`
  - 3 does not work



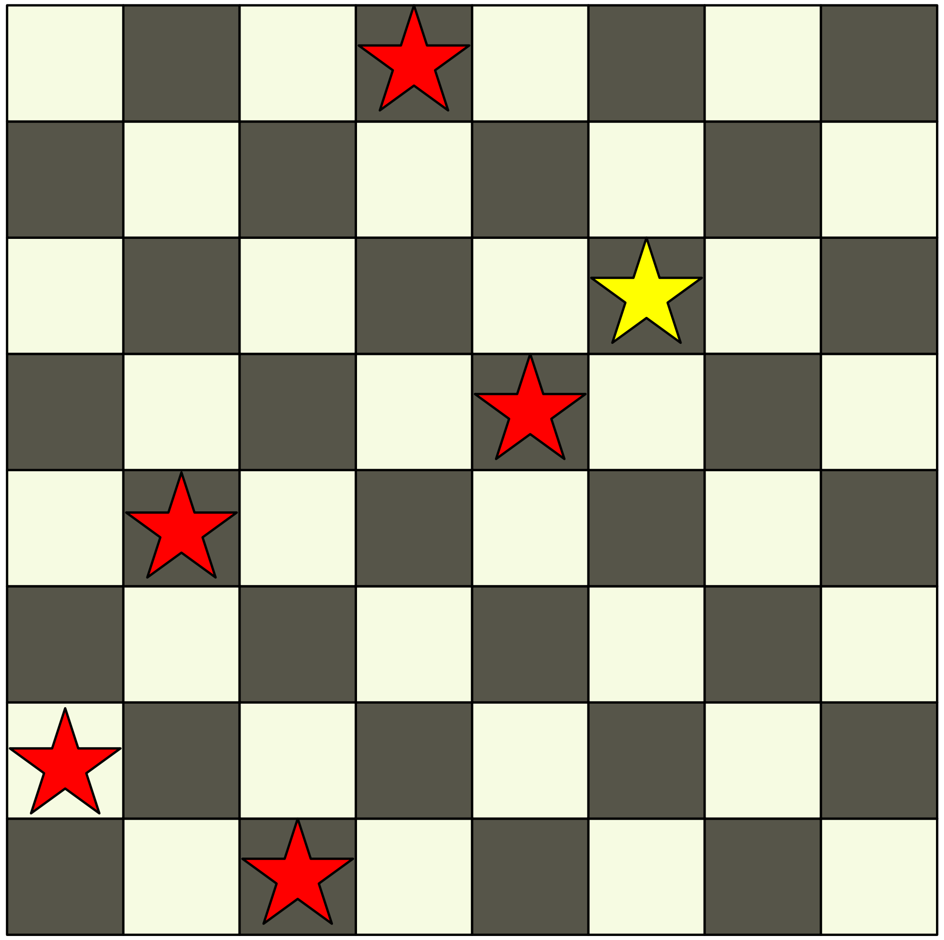
Back Tracking

- E.g.
  `board=[1,3,0,7,4,4]`
  - 4 does not work



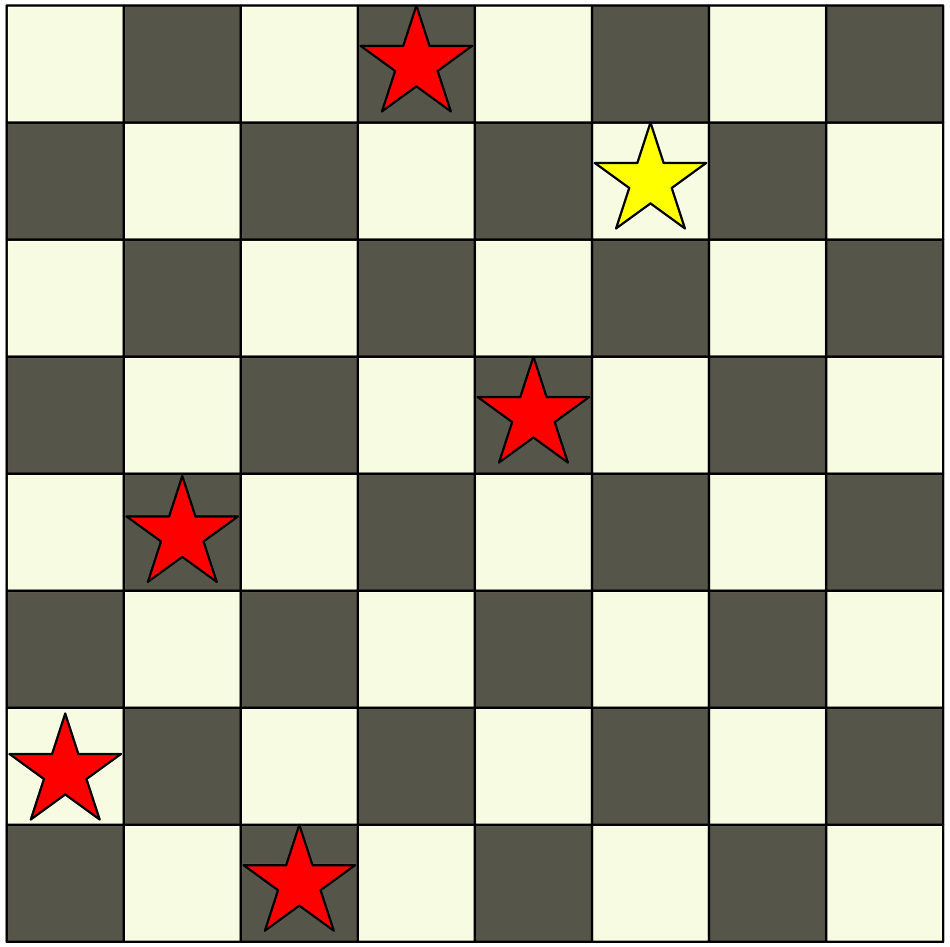
Back Tracking

- E.g.
  `board=[1,3,0,7,4,5]`
  - `5` does not work



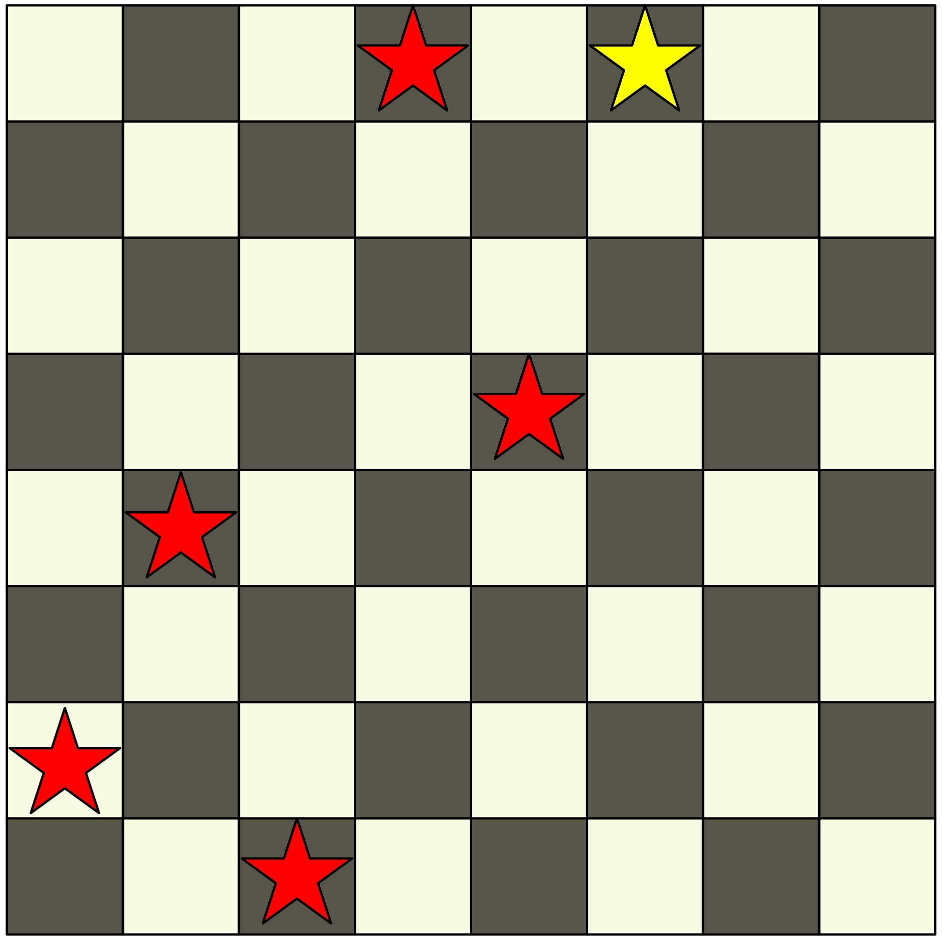
Back Tracking

- E.g.
  `board=[1,3,0,7,4,6]`
  - `6` does not work



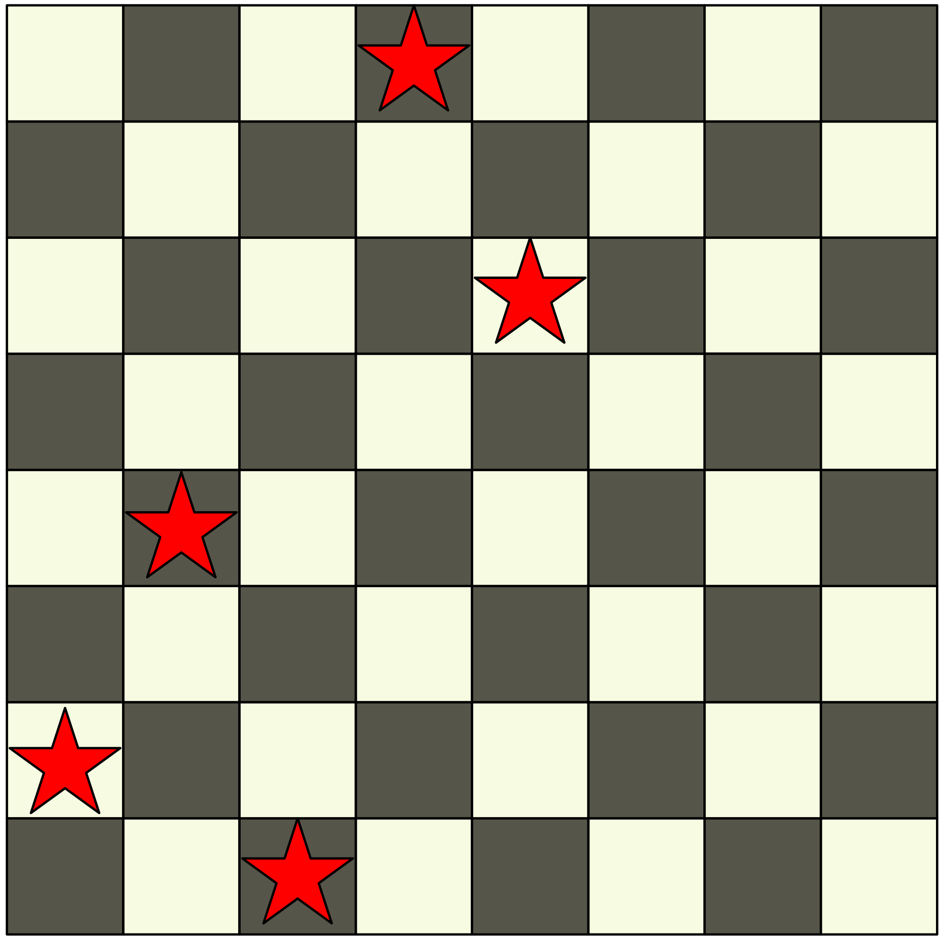
Back Tracking

- E.g.
  `board=[1,3,0,7,4,7]`
  - `7` does not work



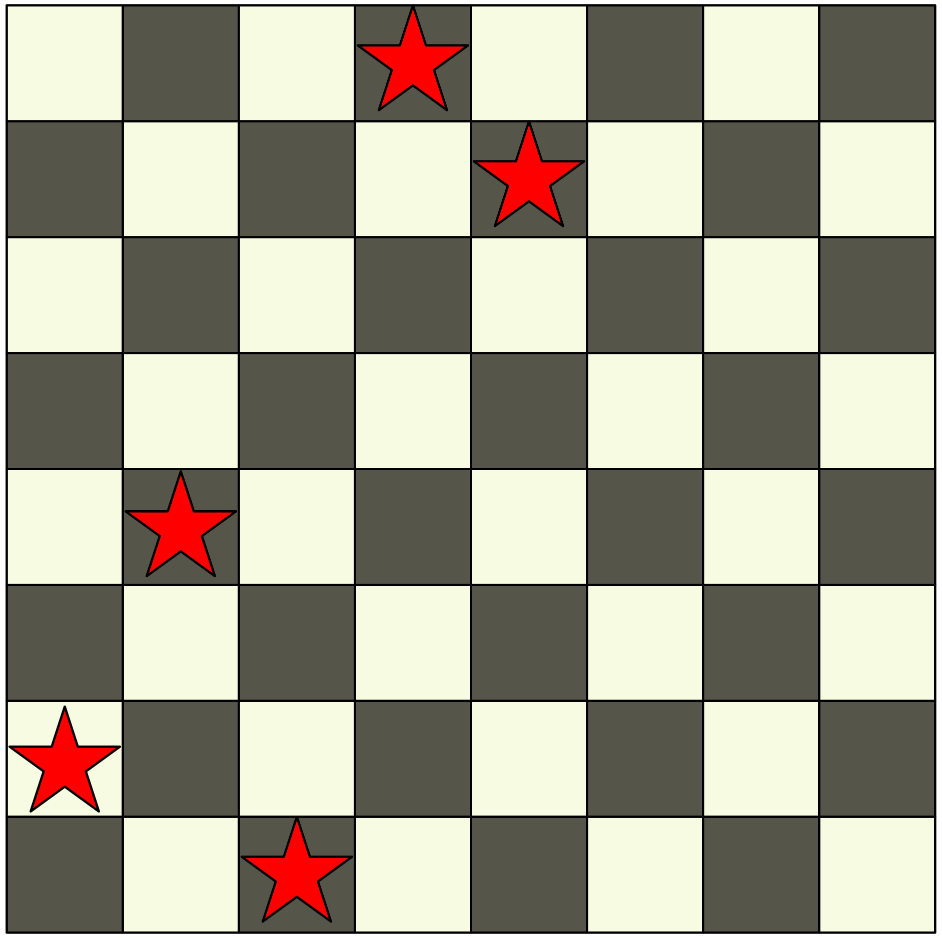
Back Tracking

- E.g. `board=[1,3,0,7,4]`
  - Since we exhausted all possibilities, we know this position is hopeless
  - So we move on to the next possibility
  - `board=[1,3,0,7,5]`
  - Which does not work



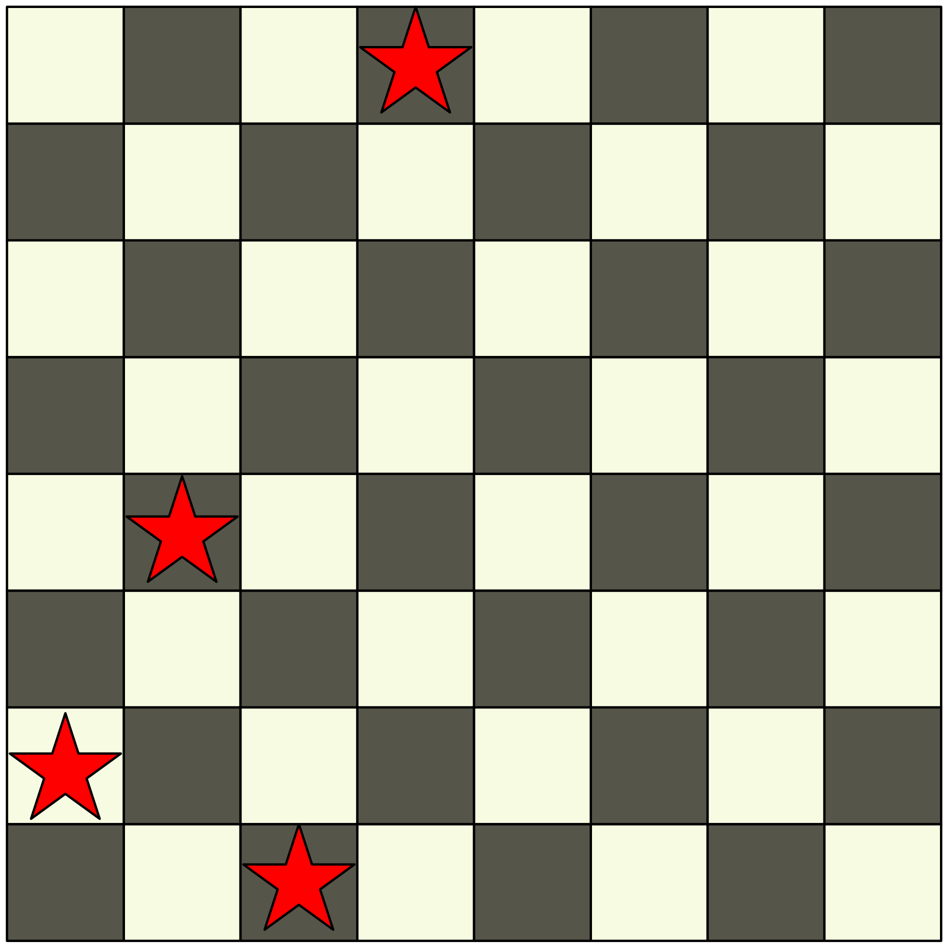
Back Tracking

- E.g. `board=[1,3,0,7,6]`
    - Not valid



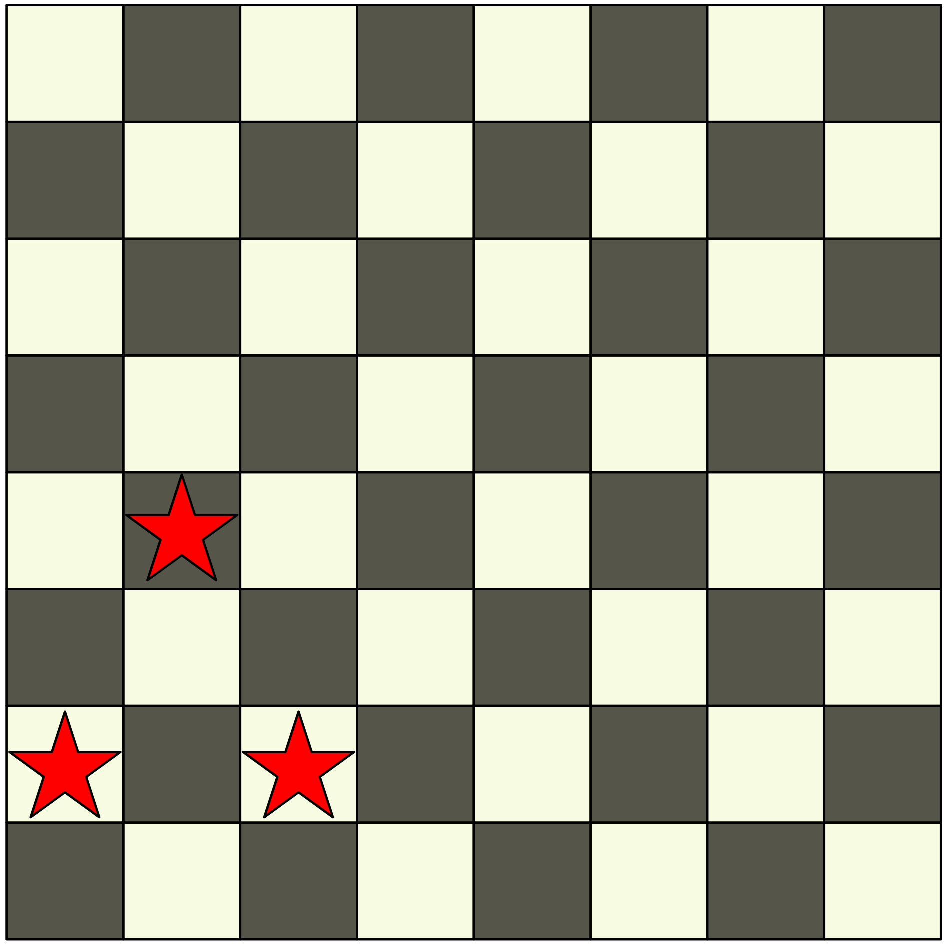
Back Tracking

- E.g. `board=[1,3,0,7]`
  - Not valid
  - So, we remove and return



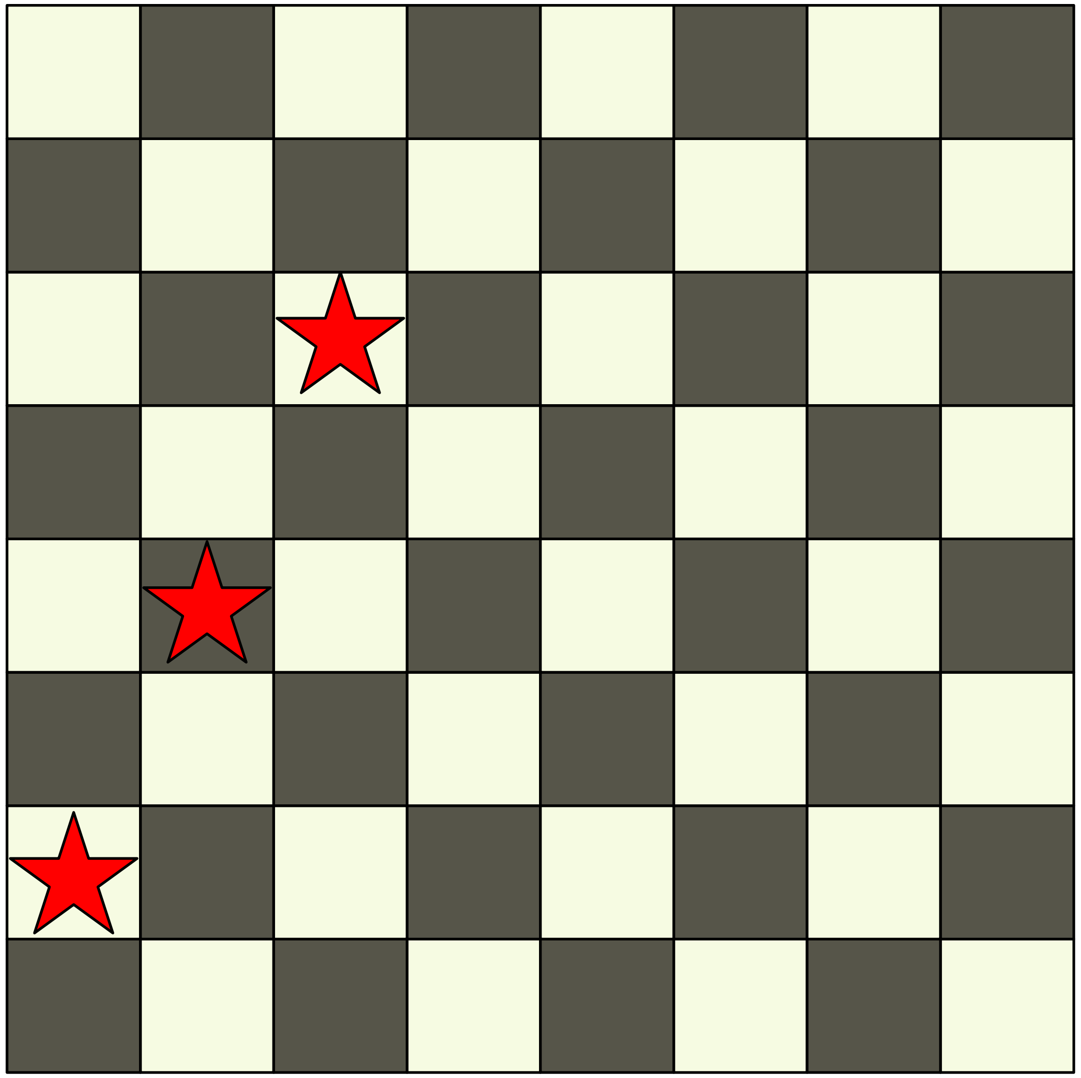
Back Tracking

- E.g. `board=[1,3,0]`
  - Now more possibilities in column 3
  - We return and board is now `[1,3]` and we try the next possibility `[1,3,1]`
  - This is invalid



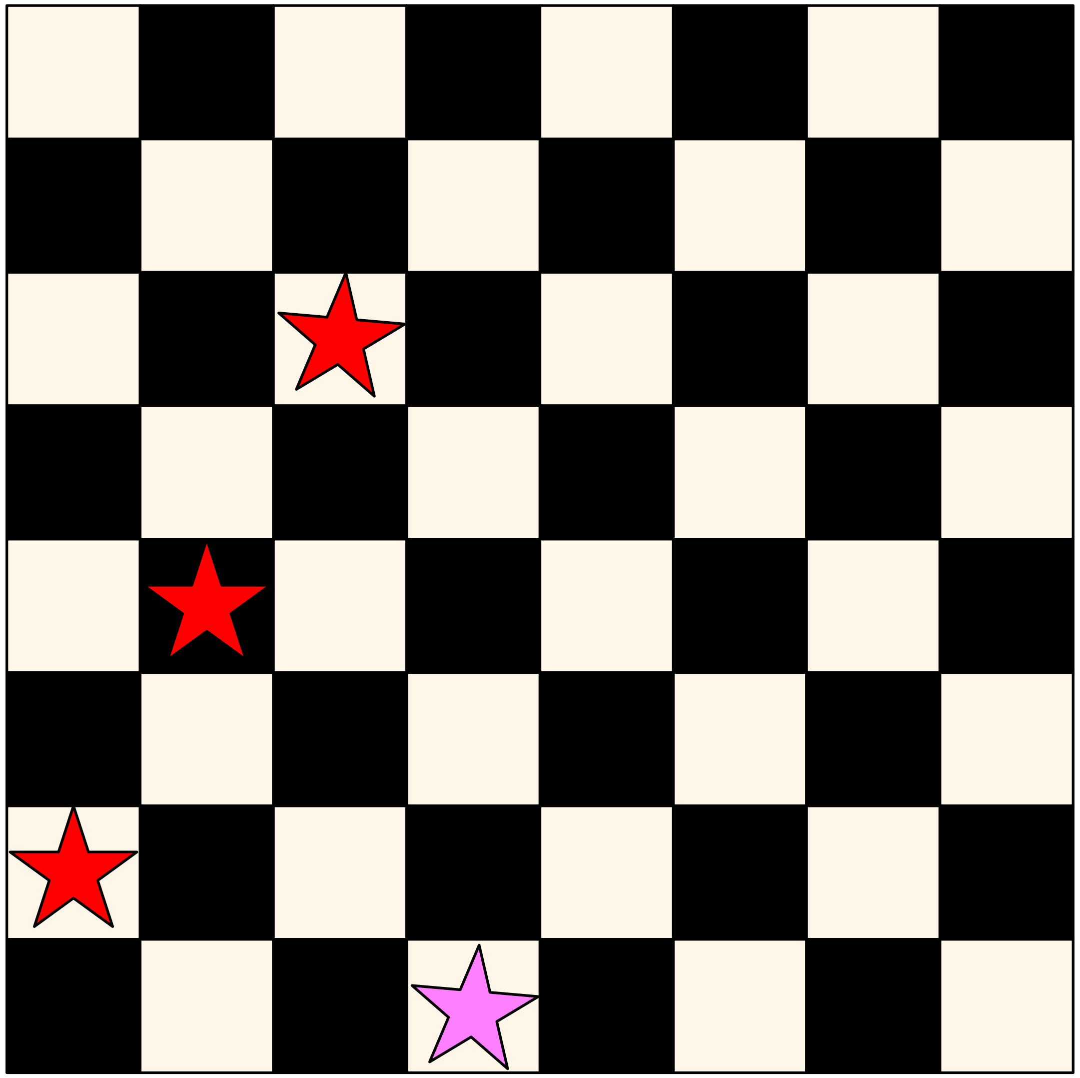
Back Tracking

- E.g. `board=[1,3]`
    - First valid partial board is
        - `board=[1,3,5]`



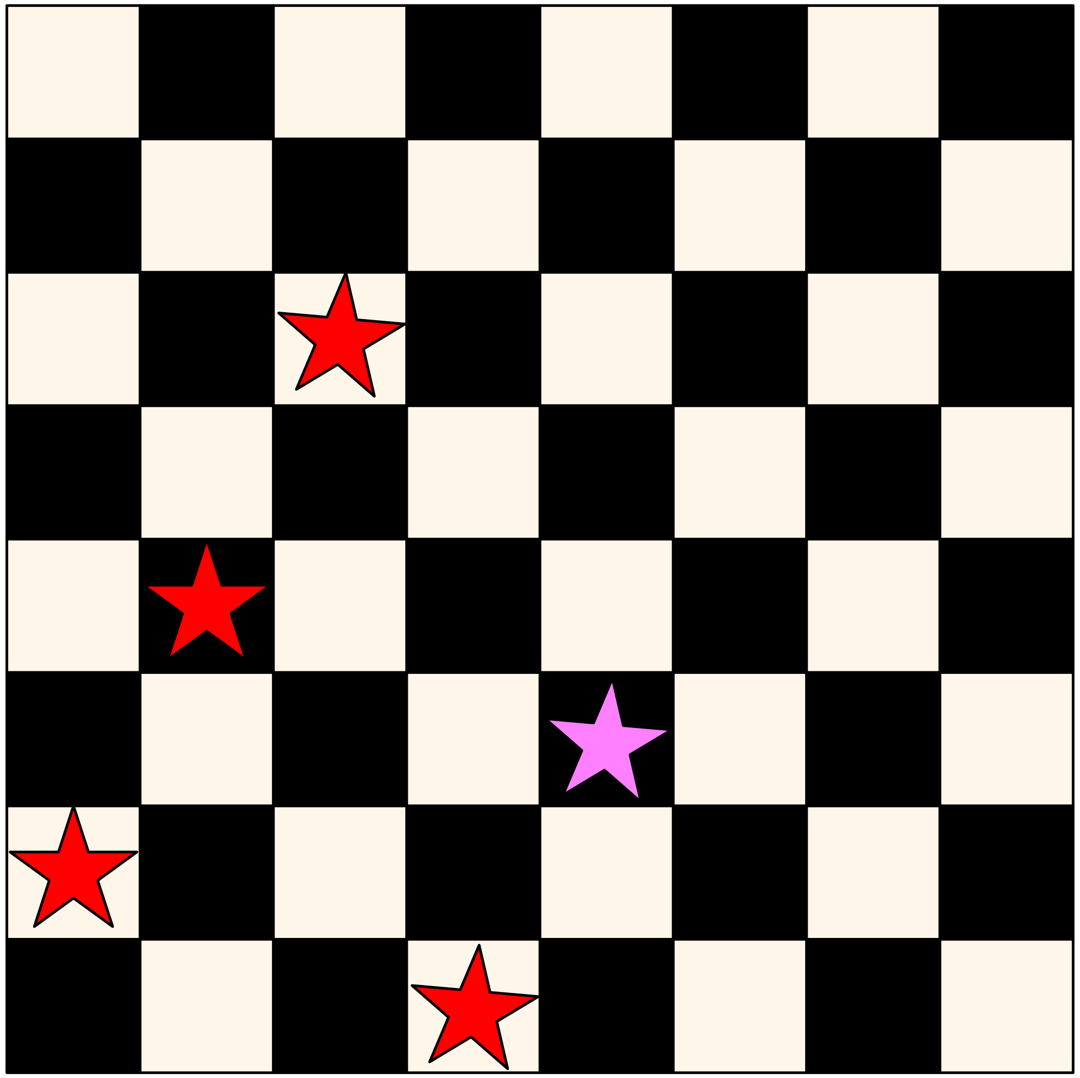
Back Tracking

- E.g. `board=[1,3,5]`
  - First choice is 0



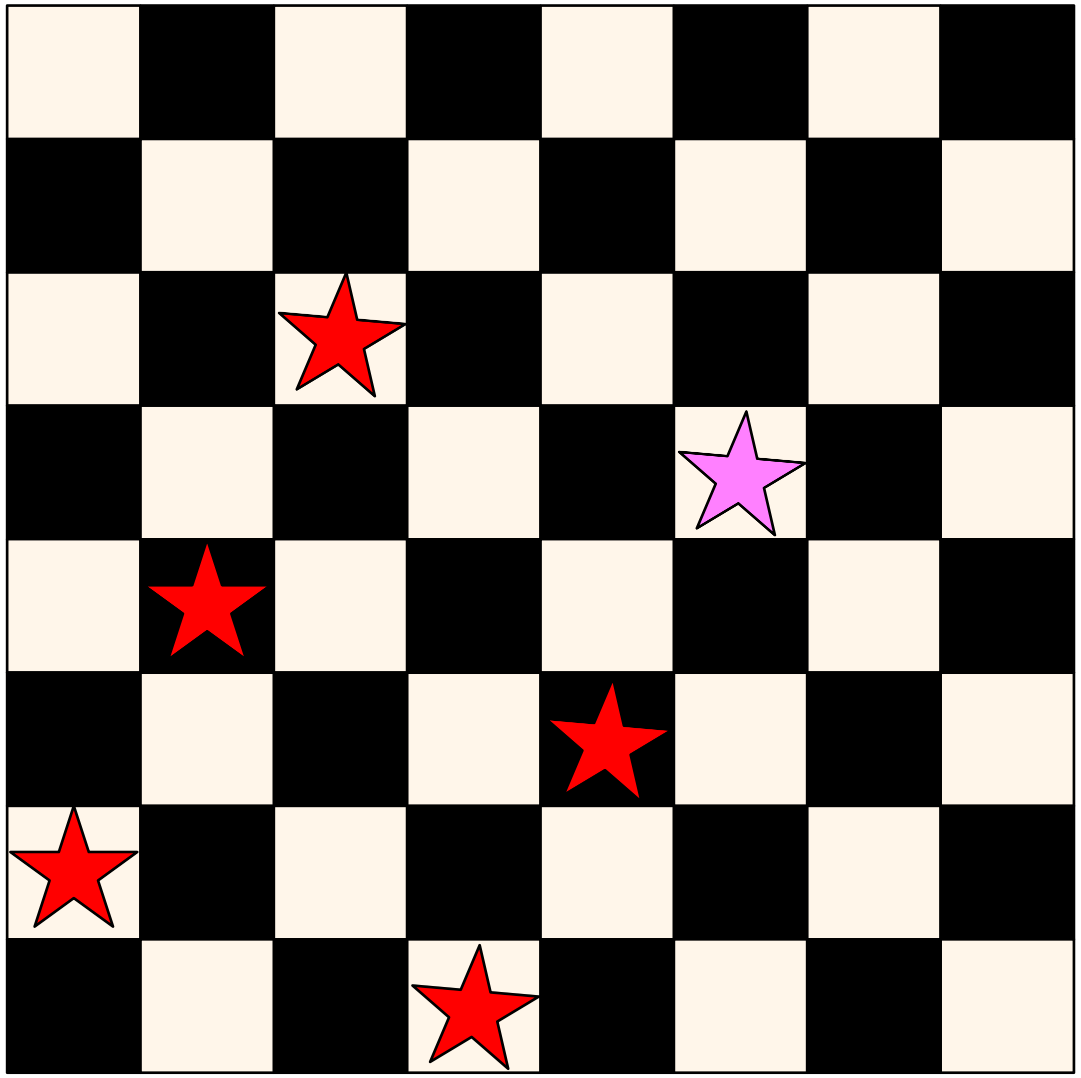
Back Tracking

- E.g. `board=[1,3,5,0]`
  - First choice is 2



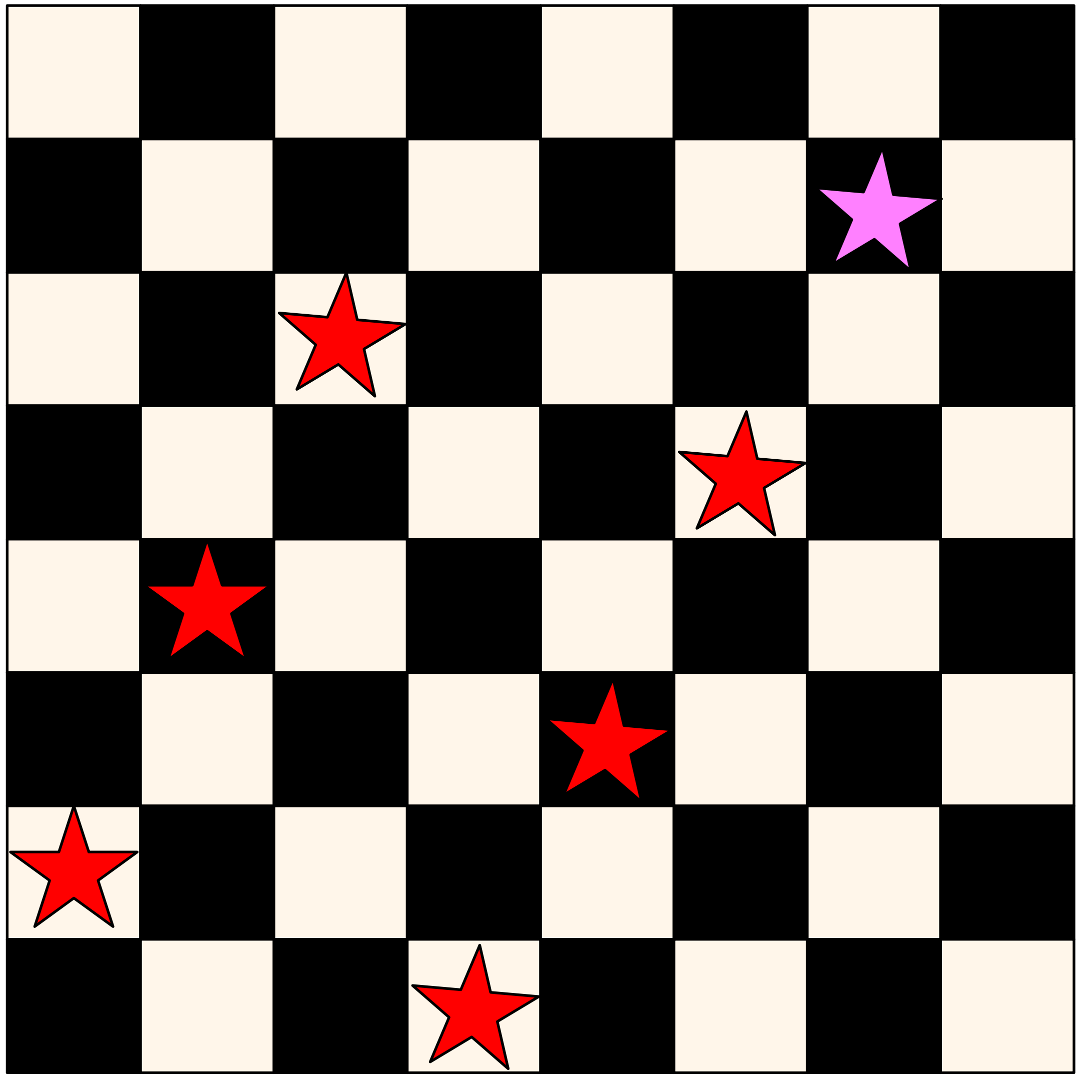
Back Tracking

- E.g. `board=[1,3,5,0,2]`
  - Select 4



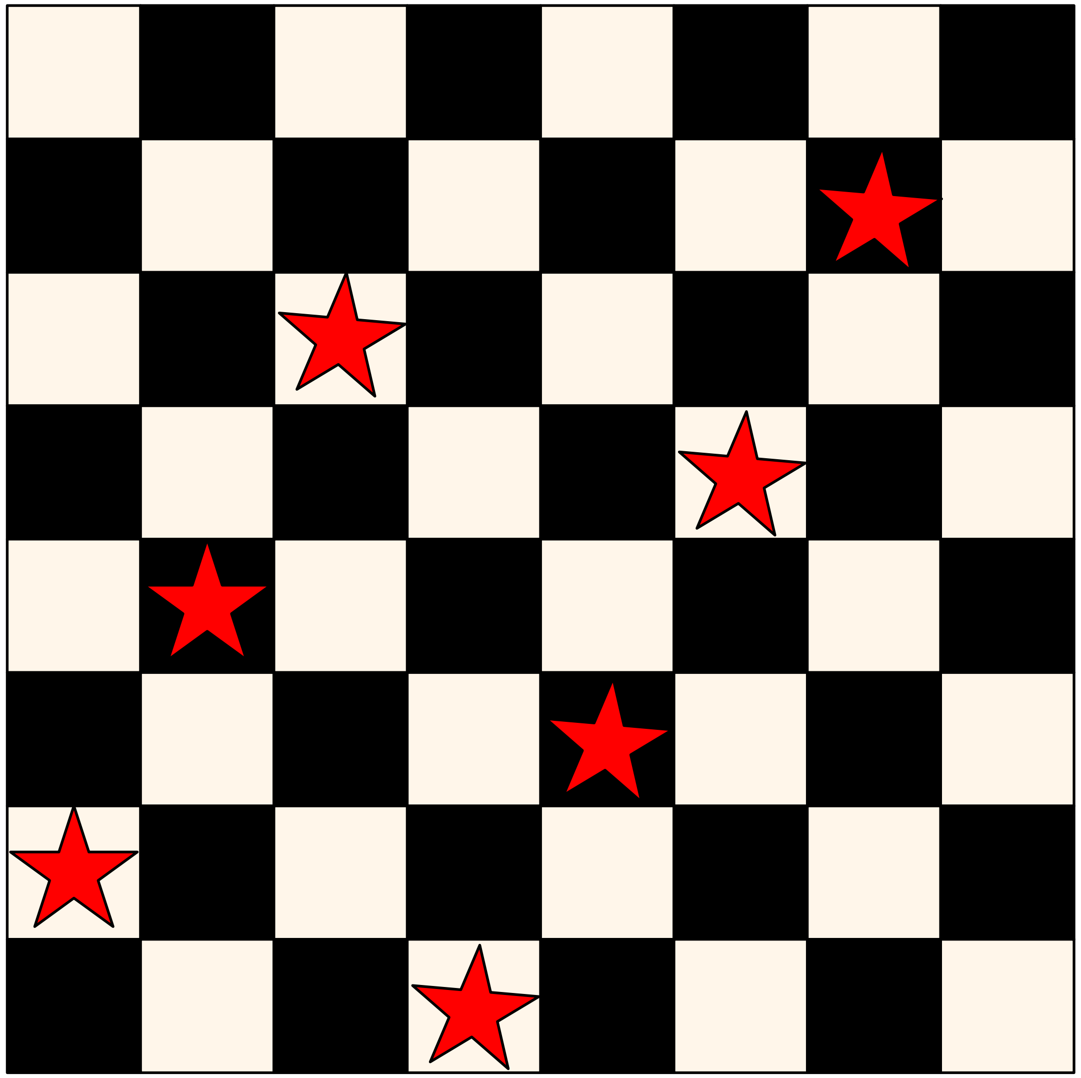
Back Tracking

- `board=[1,3,5,0,2,4]`



Back Tracking

- `[1,3,5,0,2,4,6]`
- Backtrack!

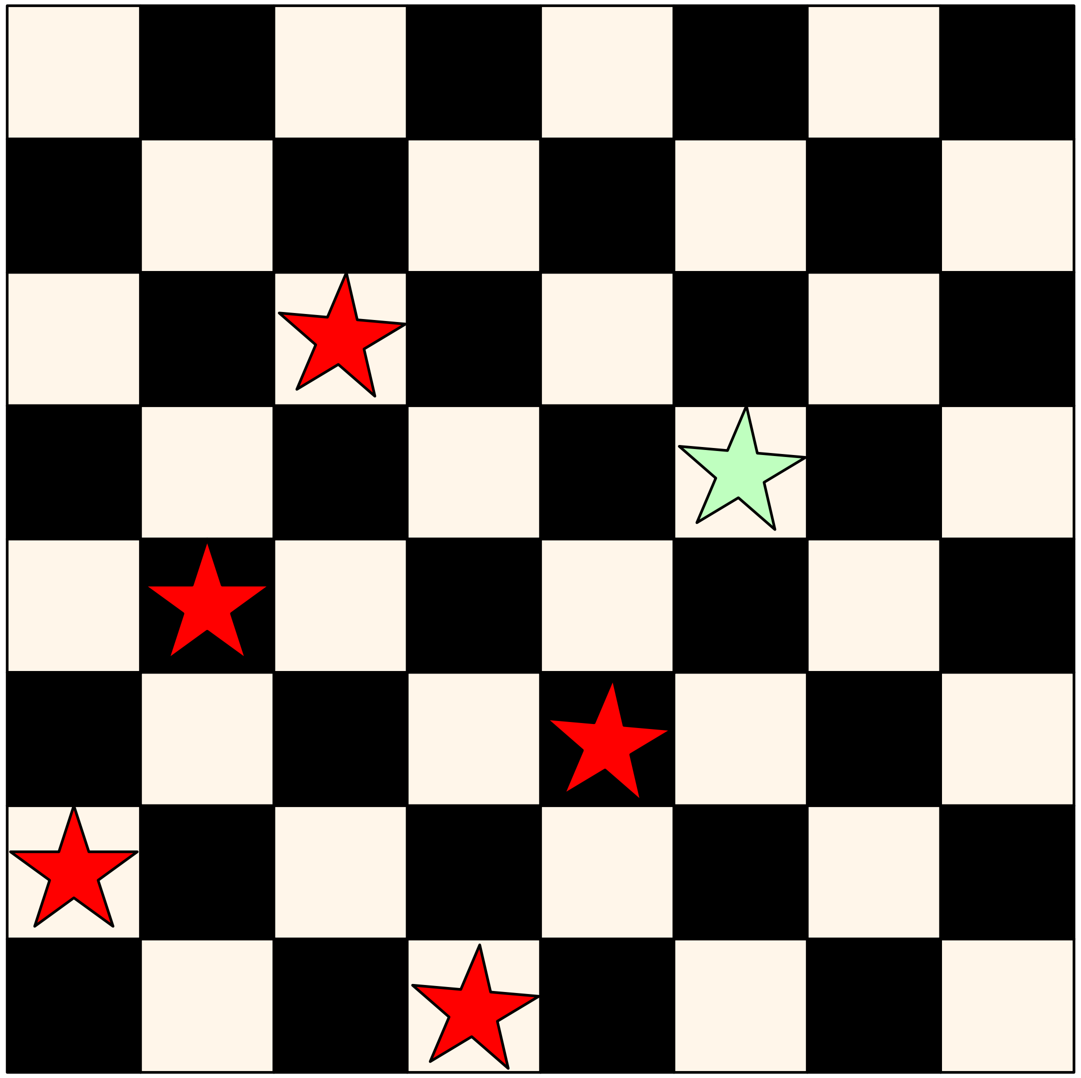


Back Tracking

- `[1,3,5,0,2,4]`
- Does not work
- Running out of options

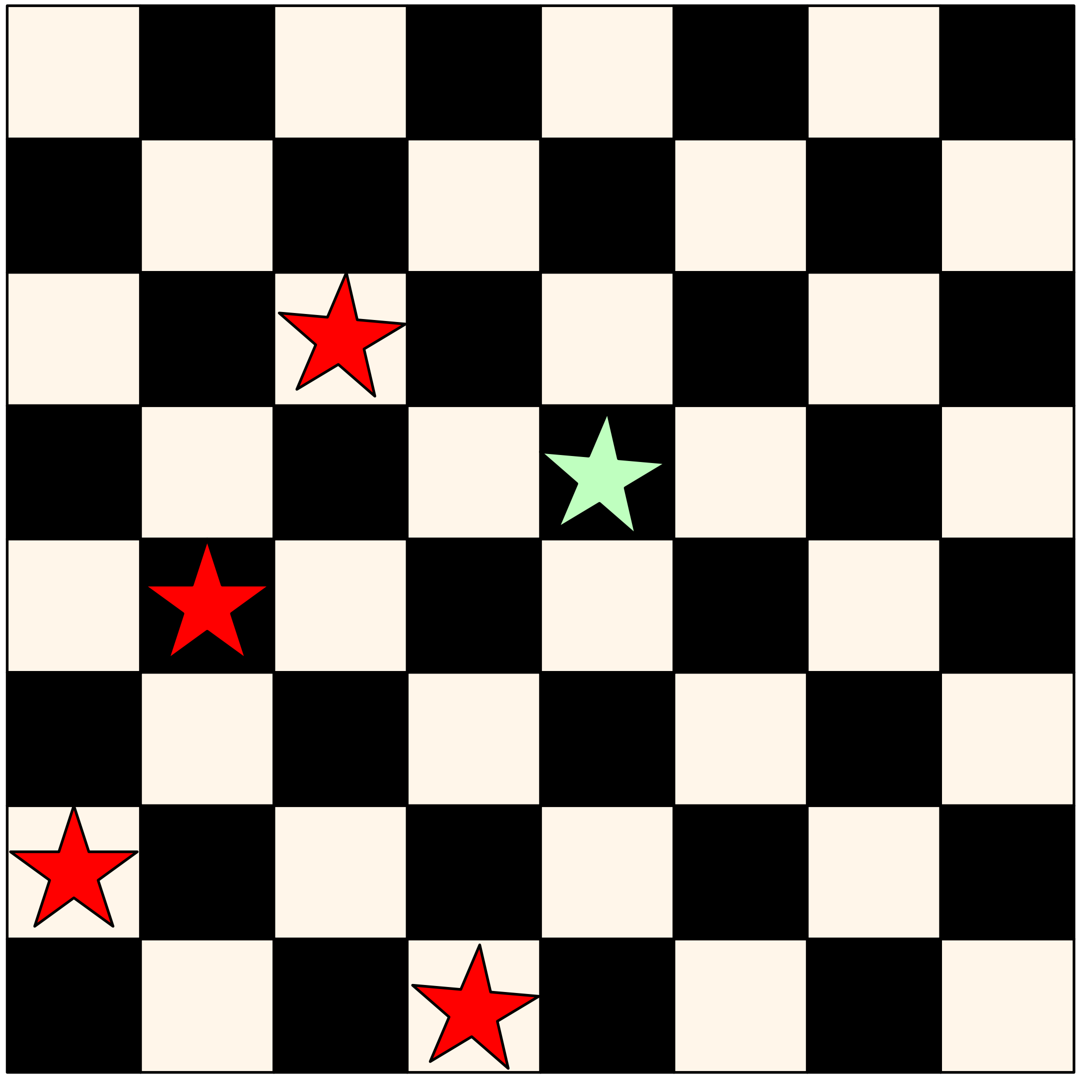
Back Tracking

- `[1,3,5,0,2]`
- Does not work
- Running out of options
- BACKTRACK!



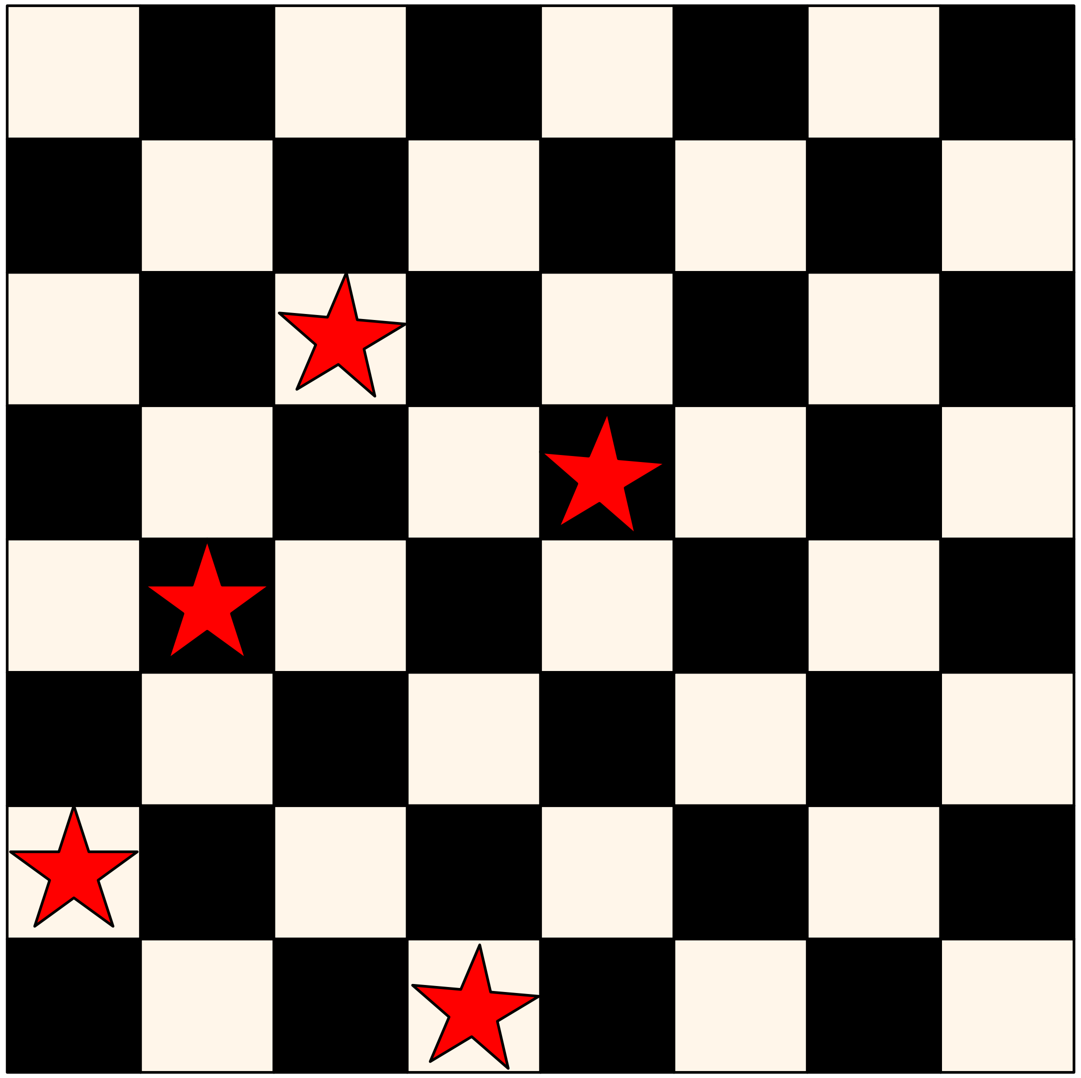
Back Tracking

- `[1,3,5,0,4]`
- Try out 4



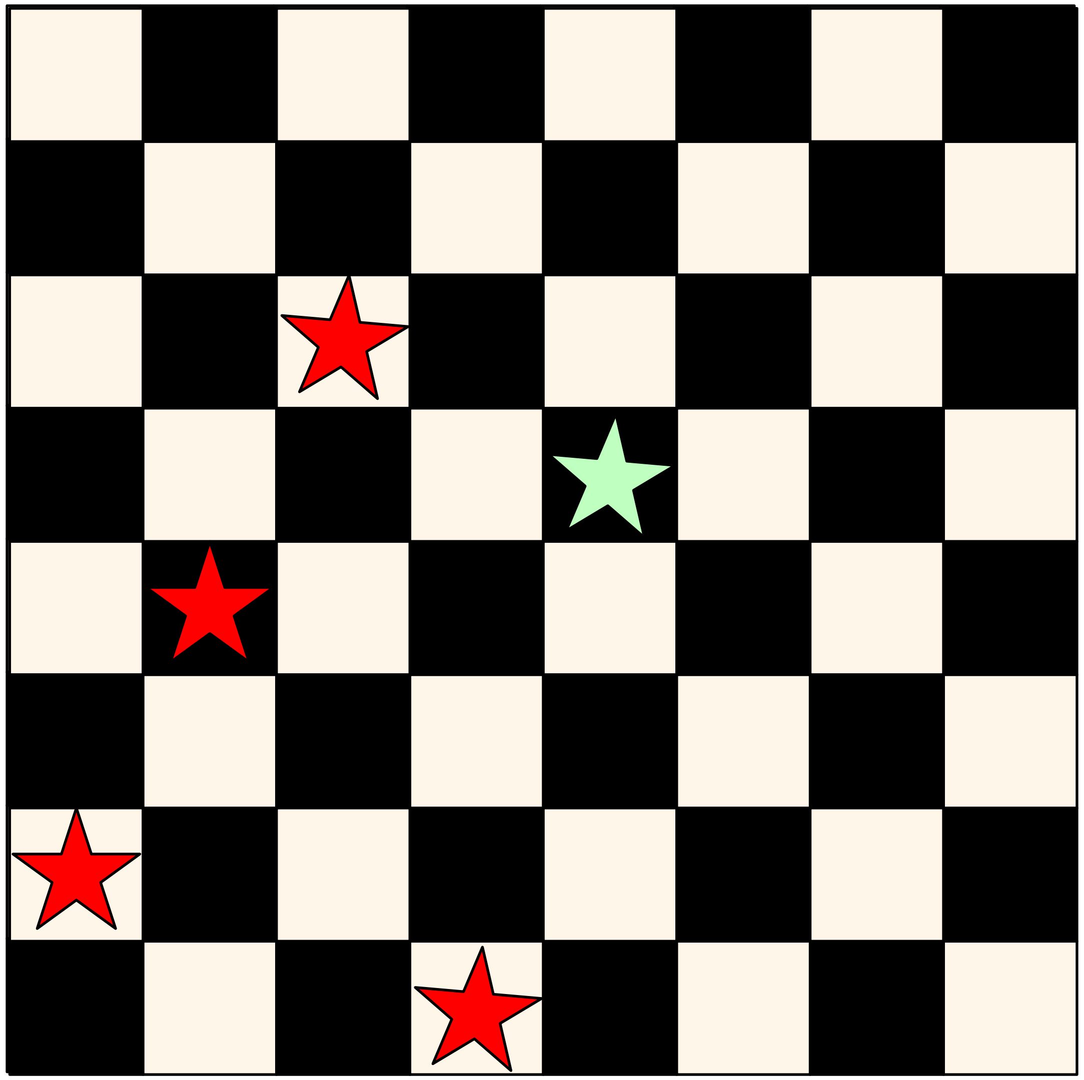
Back Tracking

- `[1,3,5,0,4]`
- No next placement
- Backtrack!



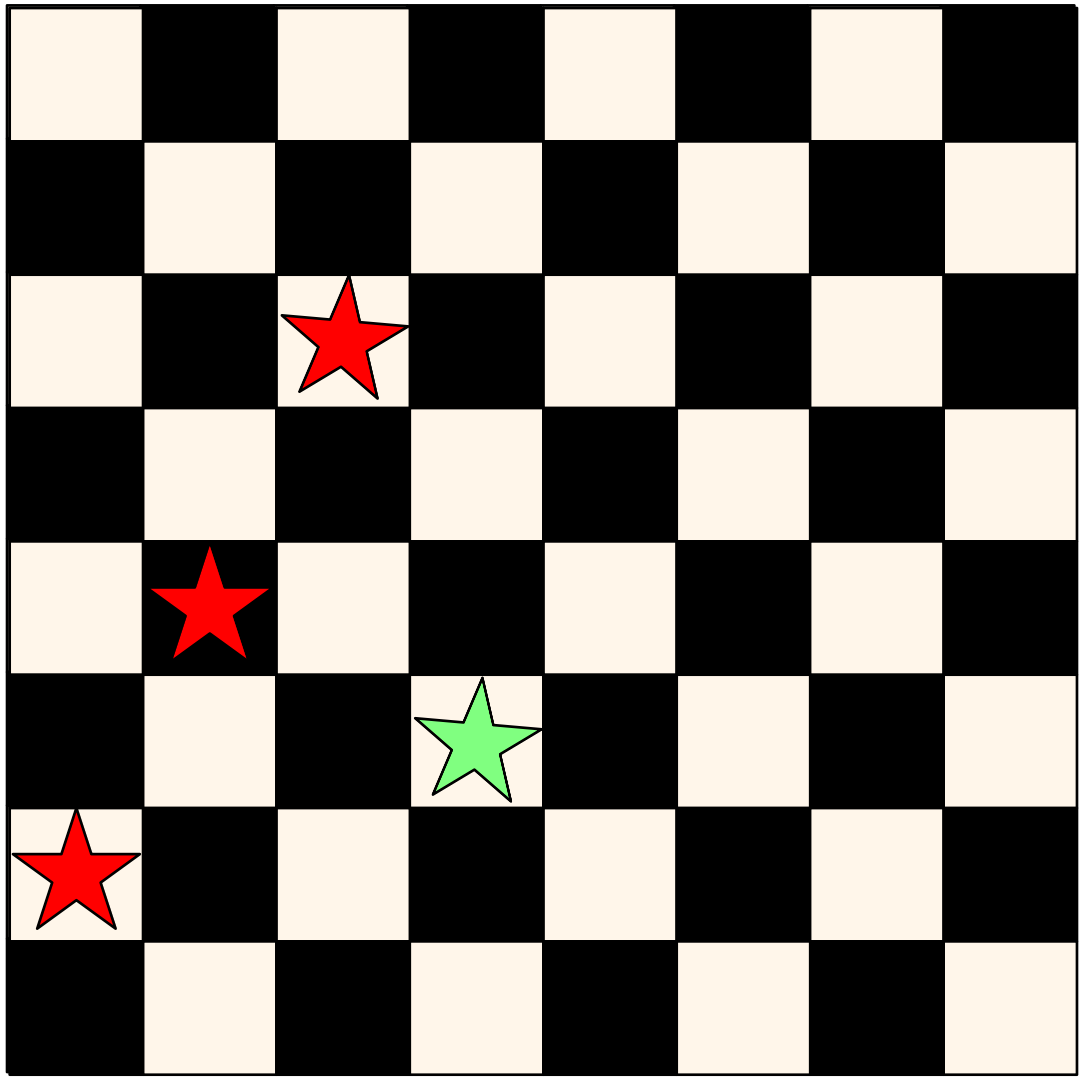
Back Tracking

- E.g. `board=[1,3,5,0]`
  - Backtrack



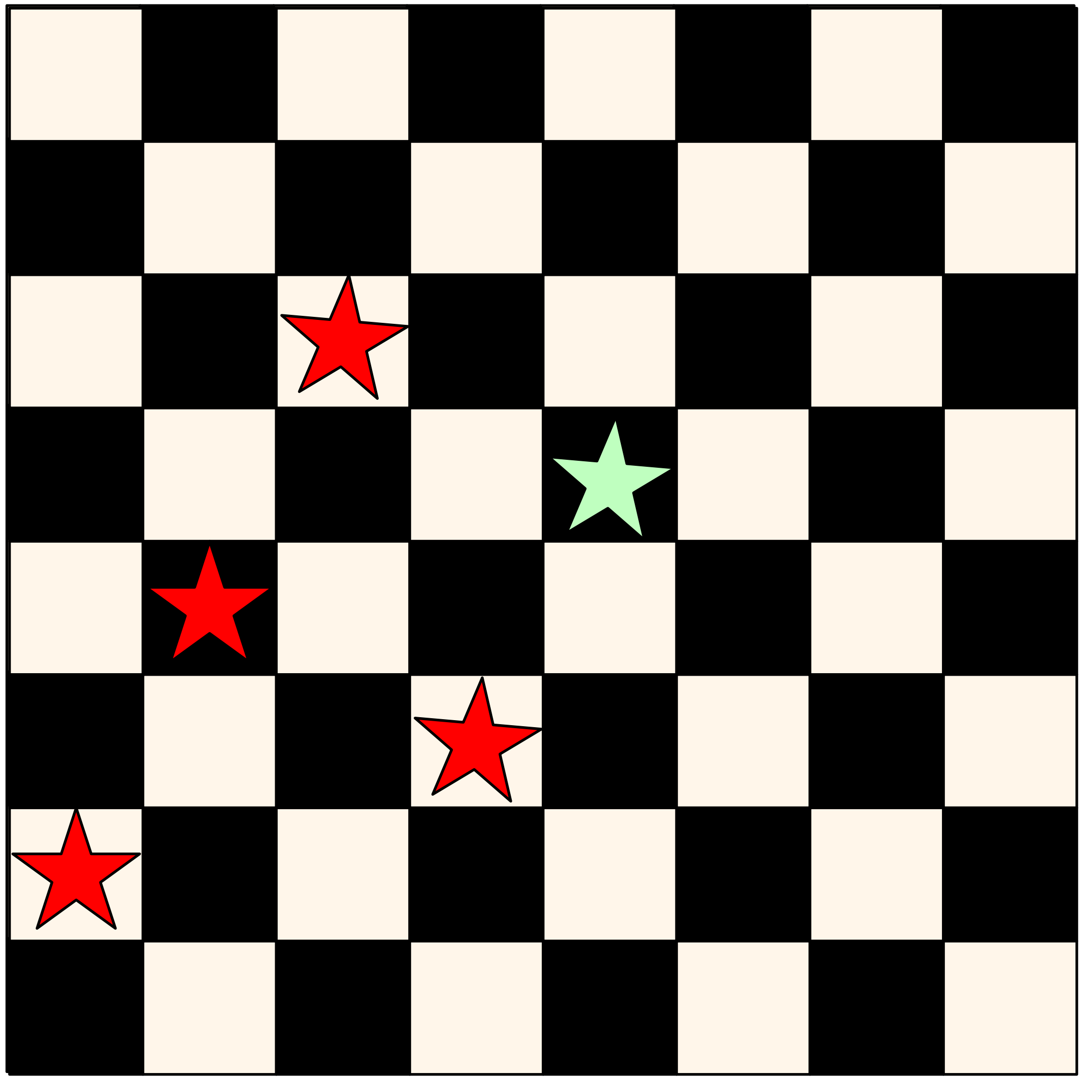
Back Tracking

- `board=[1,3,5]`
  - Select 2



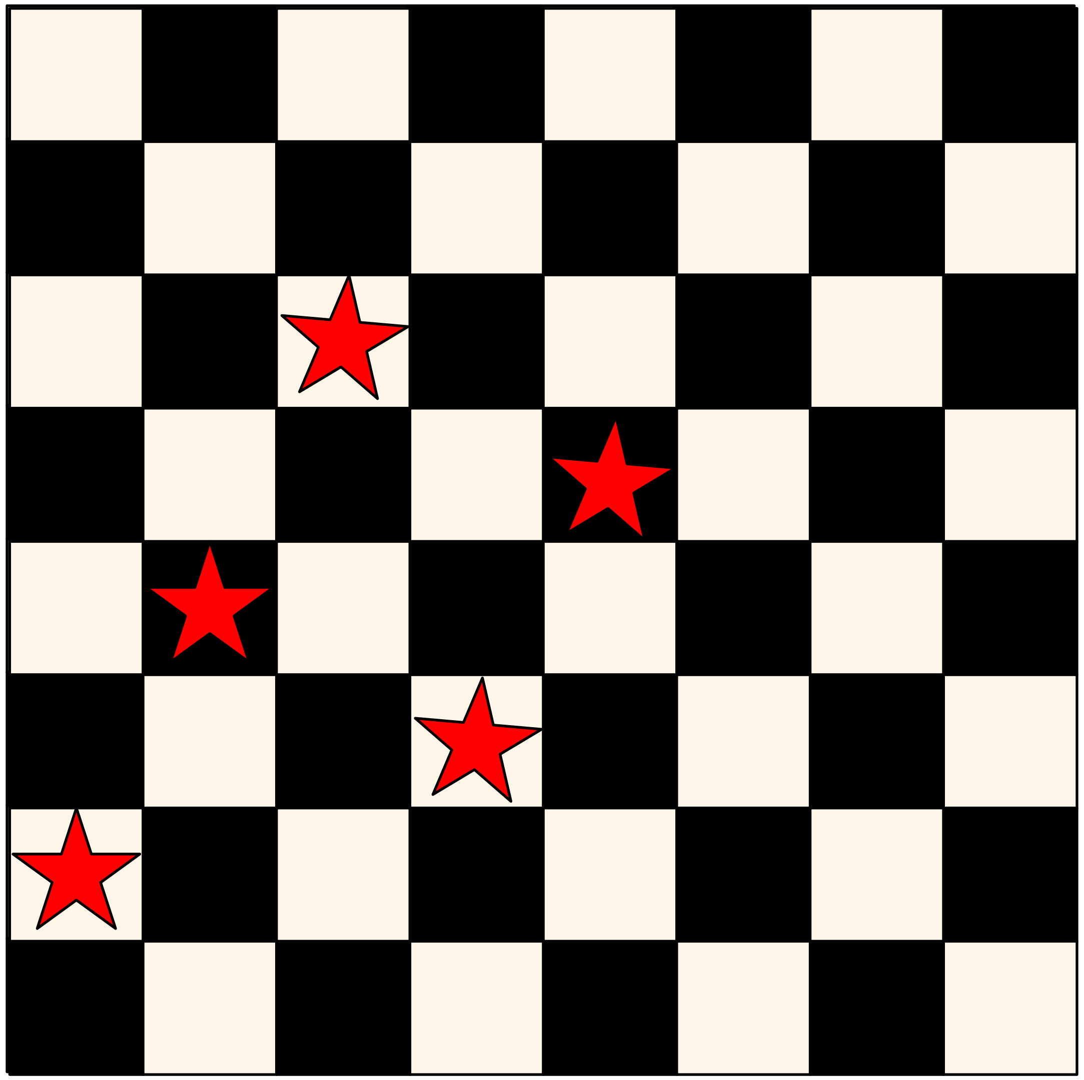
Back Tracking

- E.g. `board=[1,3,5,2]`



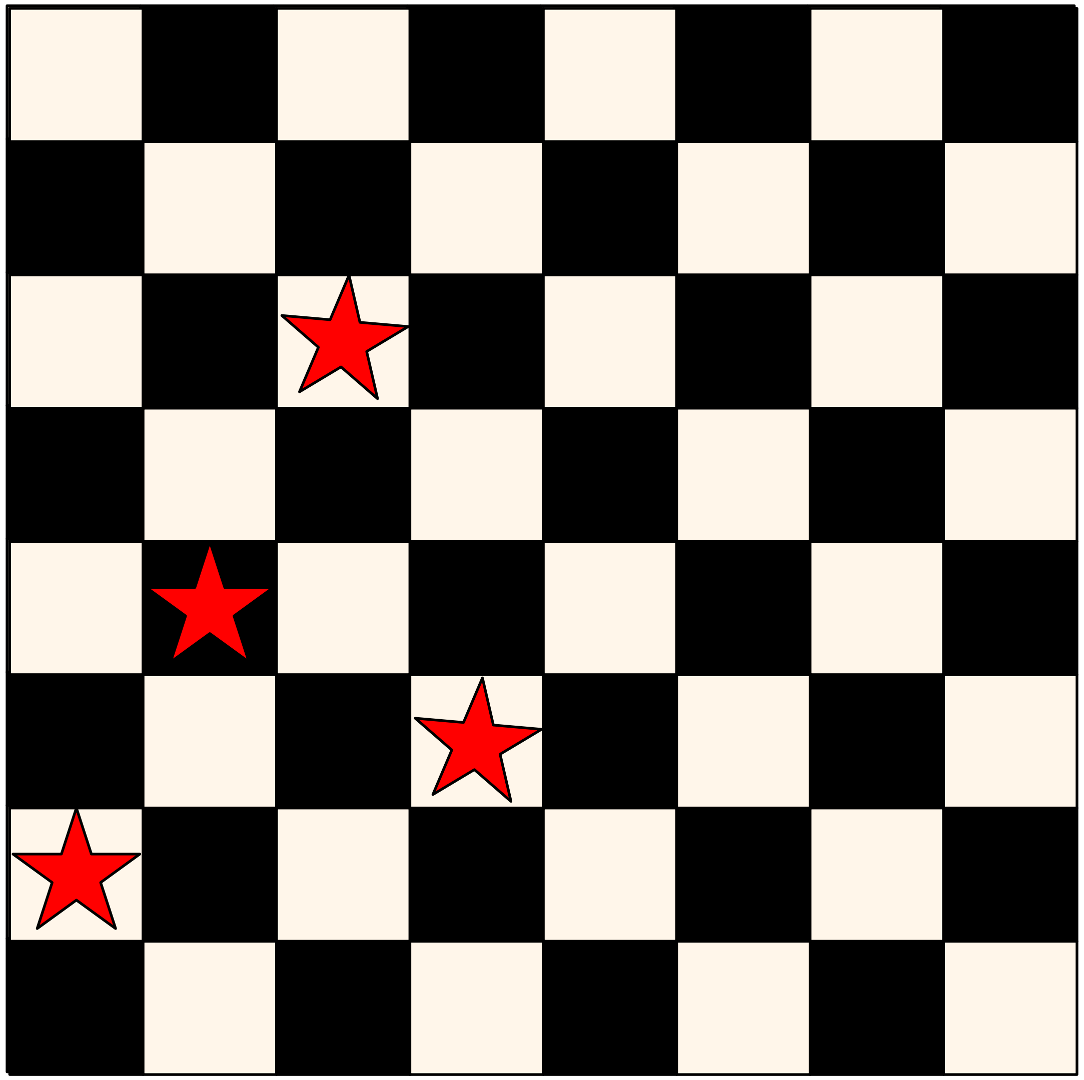
Back Tracking

- E.g. `board=[1,3,5,2,4]`
  - Backtrack



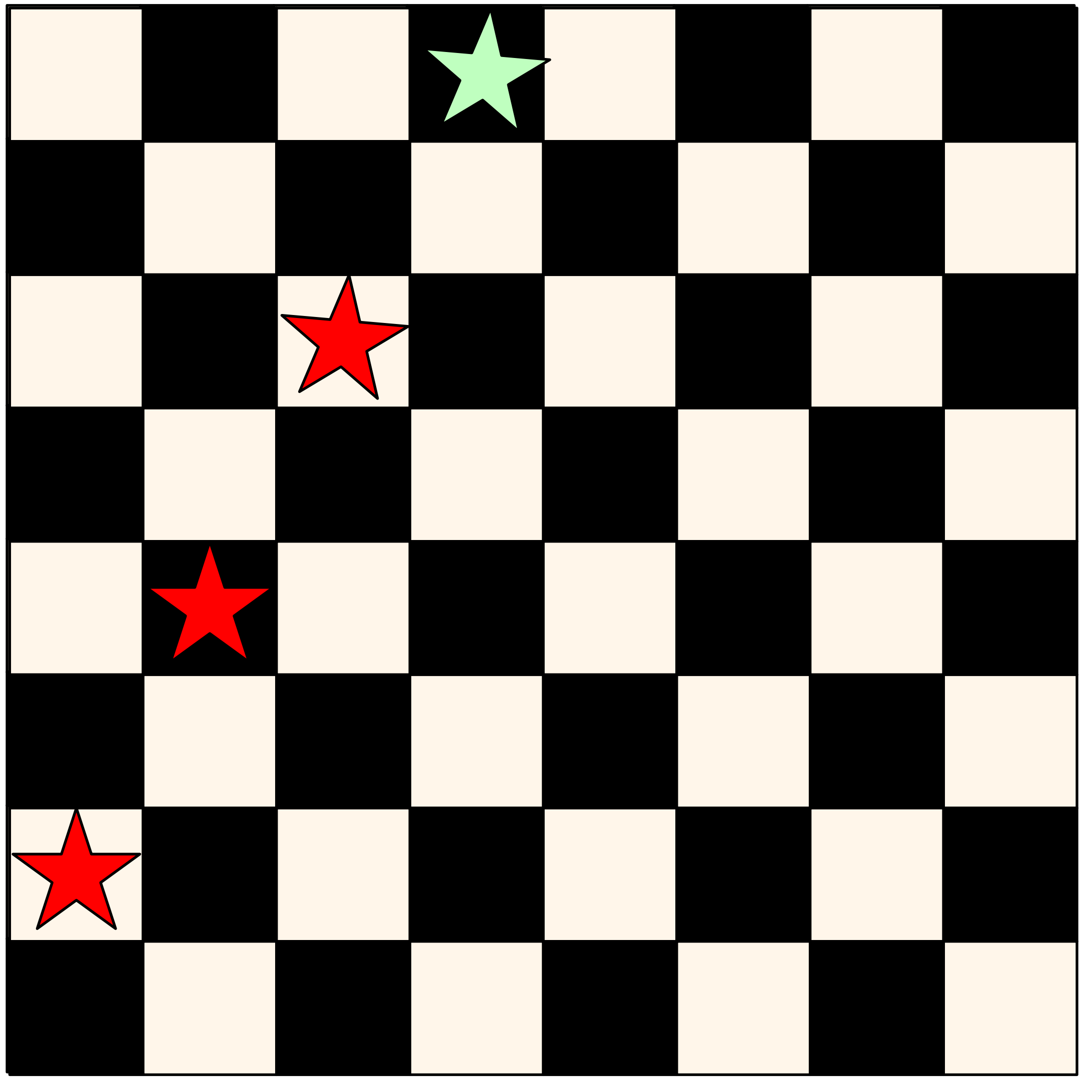
Back Tracking

- E.g. `board=[1,3,5,2]`
  - No more placement



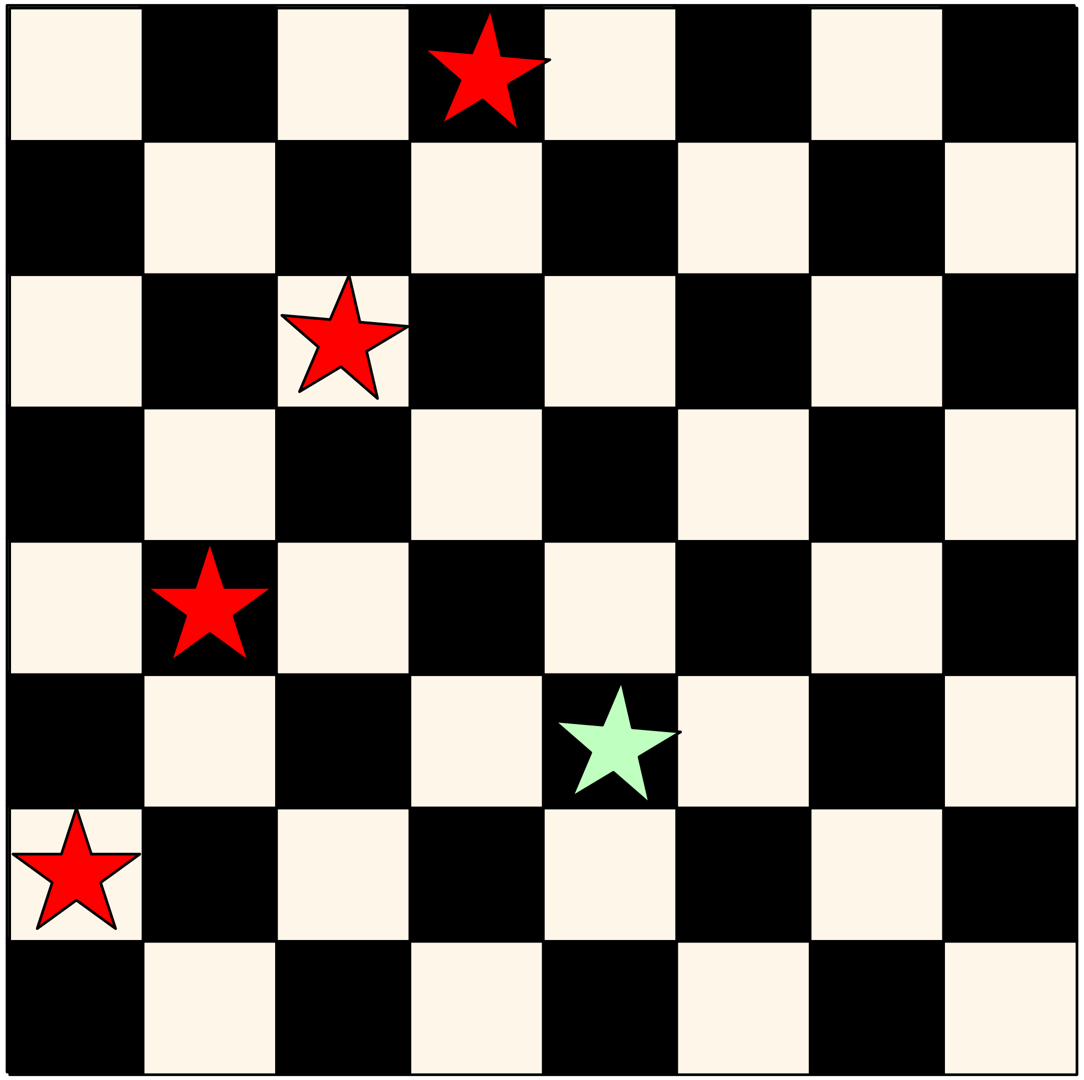
Back Tracking

- E.g. `board=[1,3,5]`
    -



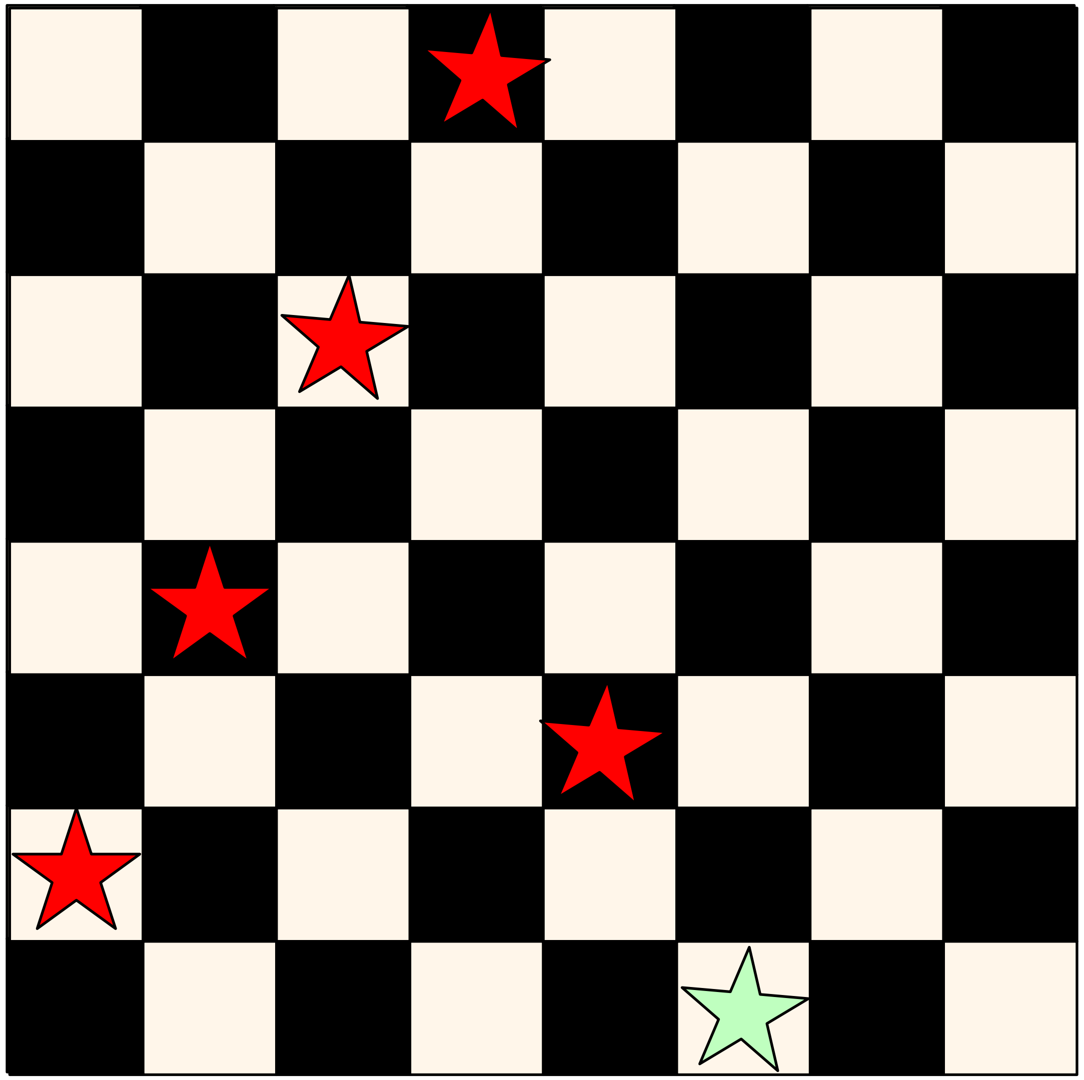
Back Tracking

- E.g. `board=[1,3,5,7]`
  -



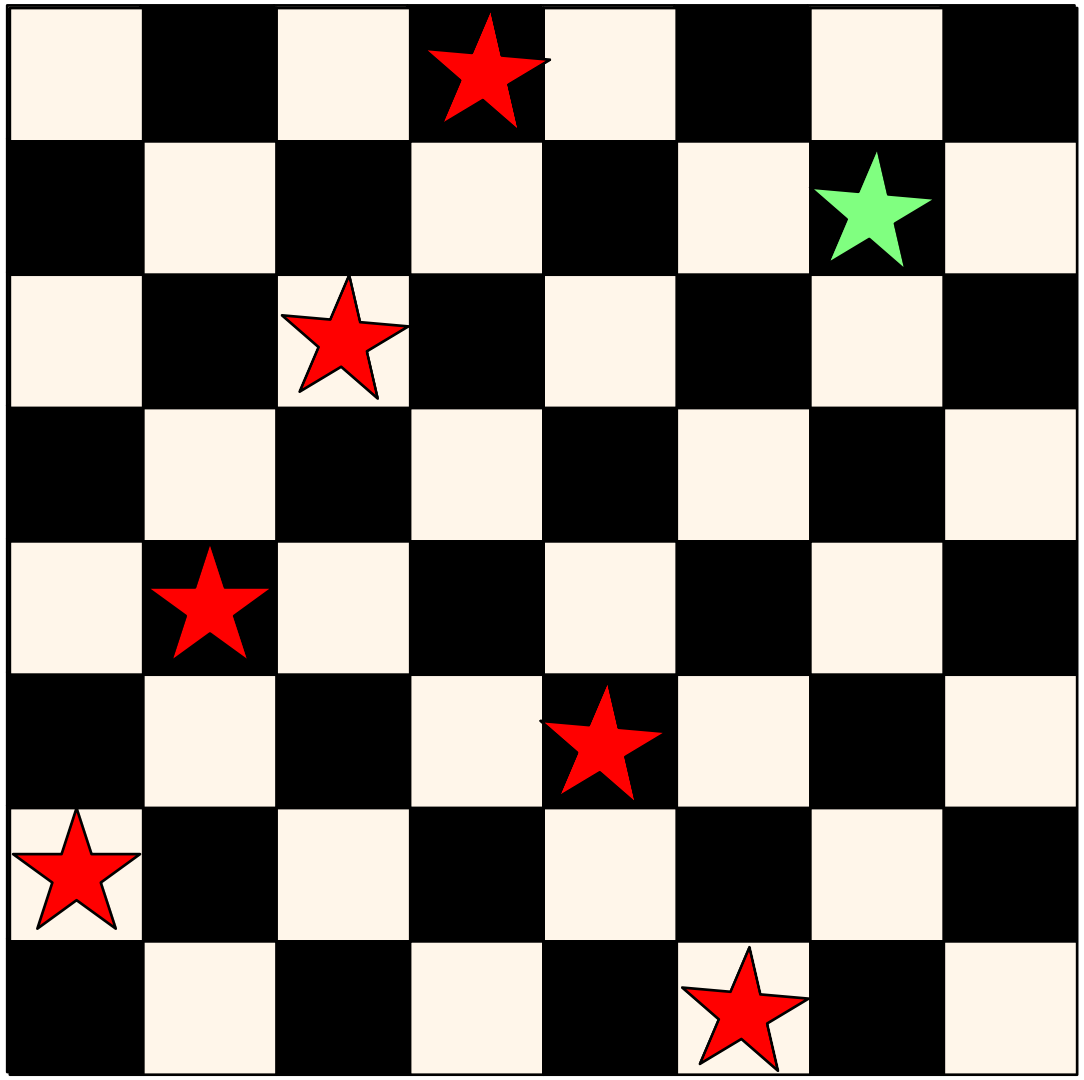
Back Tracking

- `board=[1,3,5,7,2]`



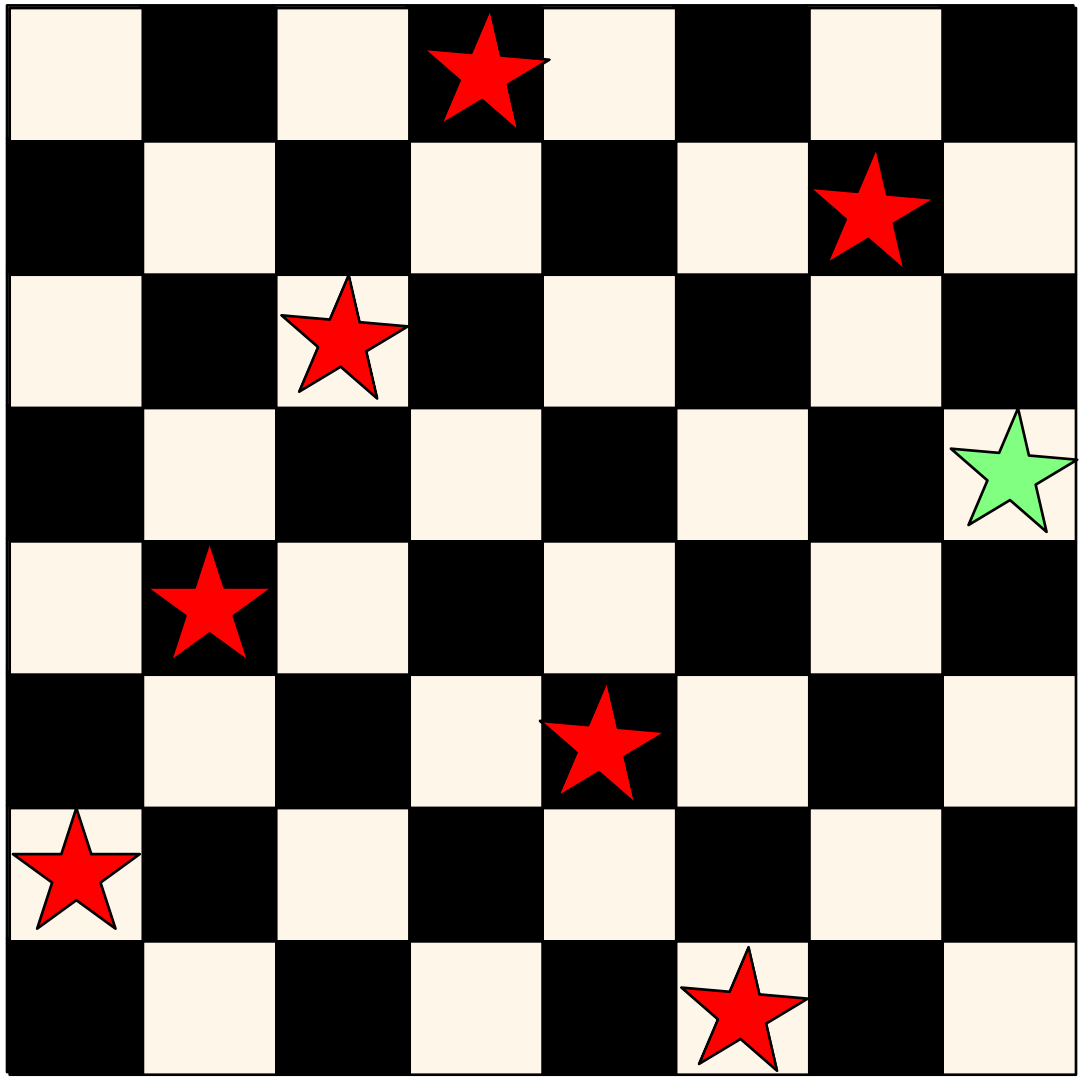
Back Tracking

- `board=[1,3,5,7,2,0]`
    -



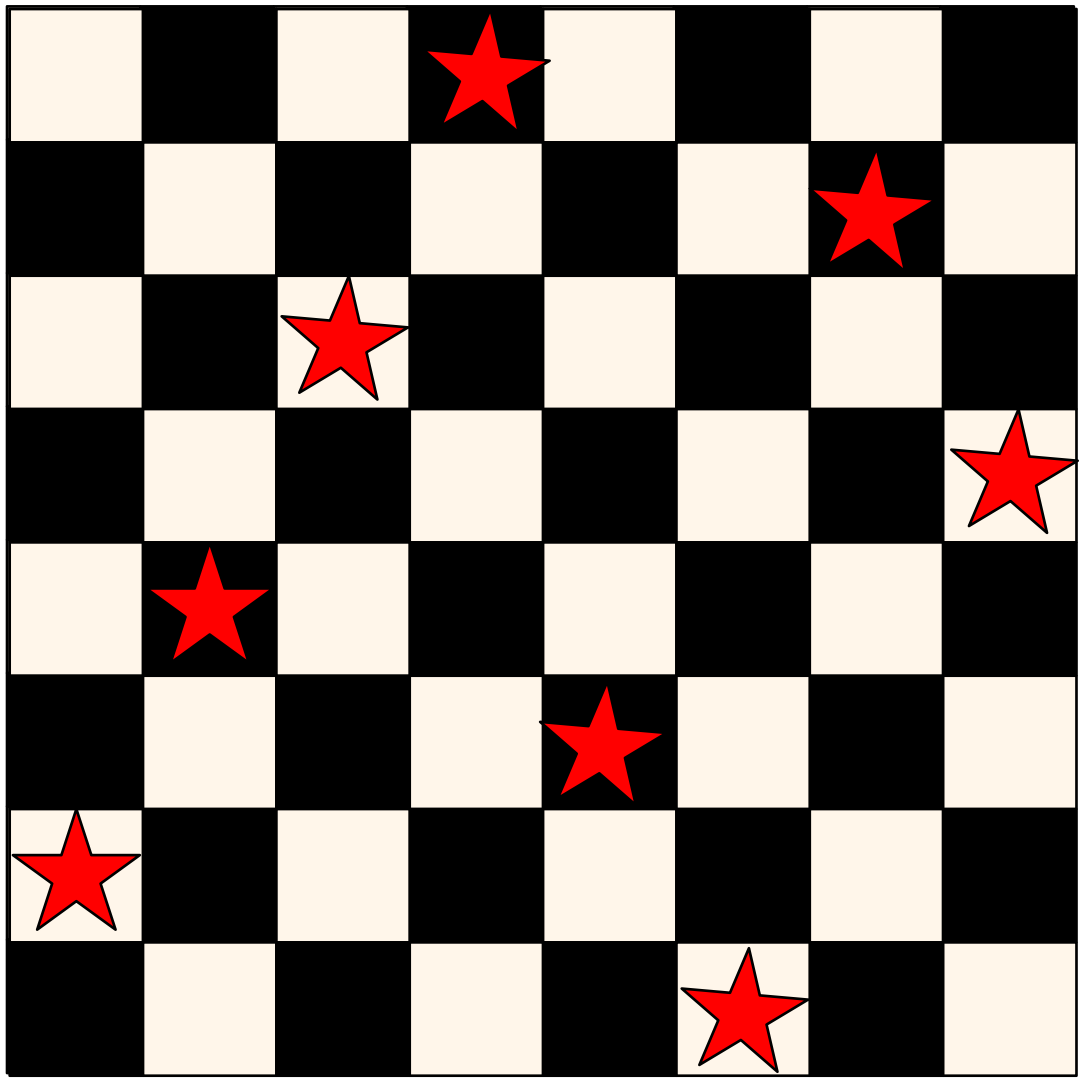
Back Tracking

- `board=[1,3,5,7,2,0]`
    -



Back Tracking

- `[1,3,5,7,2,0,6,4]`
  - Finish



Back Tracking

- Need to check validity:
    - Set-up guarantees that queens are in different columns
    - Need to check that a new queen is not in the same row or in one of the two diagonals with any already placed queen

```
def is_valid(board):
    current_queen_row, current_queen_col = len(board)-1, board[-1]
    for row, col in enumerate(board[:-1]):
        diff = abs(current_queen_col - col)
        if diff == 0 or diff == current_queen_row - row:
            return False
    return True
```

Back Tracking

```
def queens(n, board = []):
    if n == len(board):
        return board
    for col in range(n):
        board.append(col)
        if is_valid(board):
            board = queens(n, board)
            if is_valid(board) and len(board)==n:
                return (board)
        board.pop()
    return board
```

Back Tracking

- Notice how we add and a remove a value from the board

```
def queens(n, board = []):
    if n == len(board):
        return board
    for col in range(n):
        board.append(col)
        if is_valid(board):
            board = queens(n, board)
            if is_valid(board) and len(board)==n:
                return (board)
        board.pop()
    return board
```

Back Tracking

- Back-tracking can be used if
  - We can construct partial solutions
  - We can verify that a partial solution is invalid
  - Can we verify if the solution is complete

Back Tracking

- Back-tracking can be used if
  - We can construct partial solutions
  - We can verify that a partial solution is invalid
  - Can we verify if the solution is complete

Back Tracking

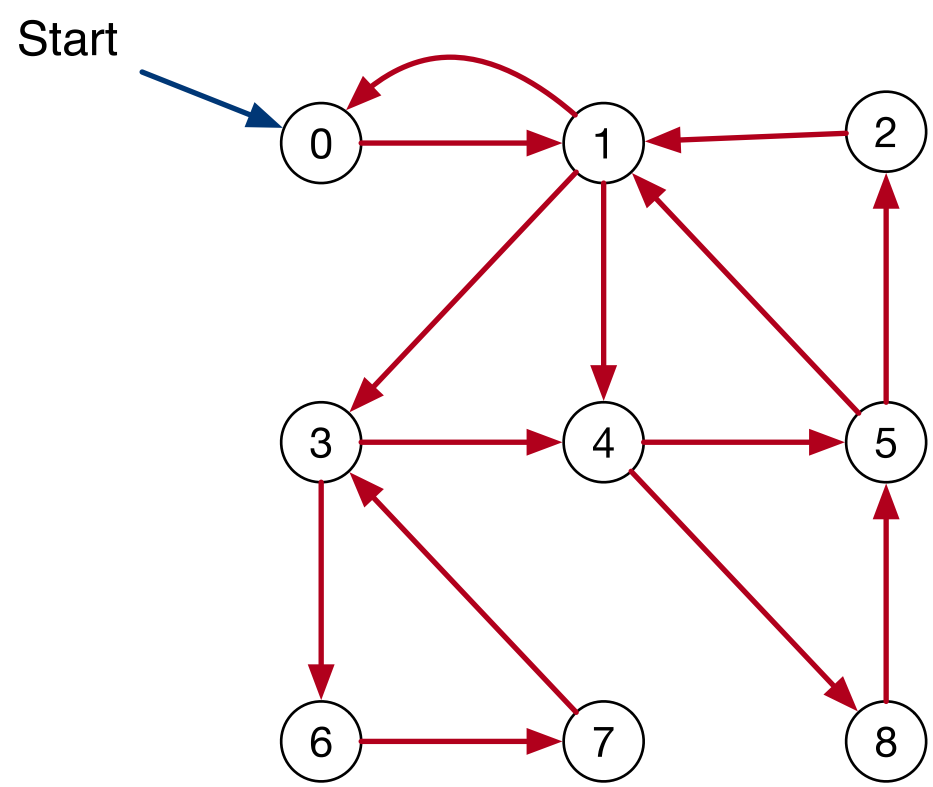
- n queens problem:
  - Can we construct partial solutions?
    - Yes, just use partial boards
  - Can we verify that a partial solution is invalid
    - Yes, if a queen is in the same row or in the same diagonal with one placed before
  - Can we verify if the solution is complete
    - Yes, when we have reached a board of length *n*.

Back Tracking

- Example: Sudoku Solver
  - Given an initial sudoku position
    - Add one new number at a time
    - Check whether that number violates any of the rules
    - Finish when all numbers have been placed

In Class Exercise

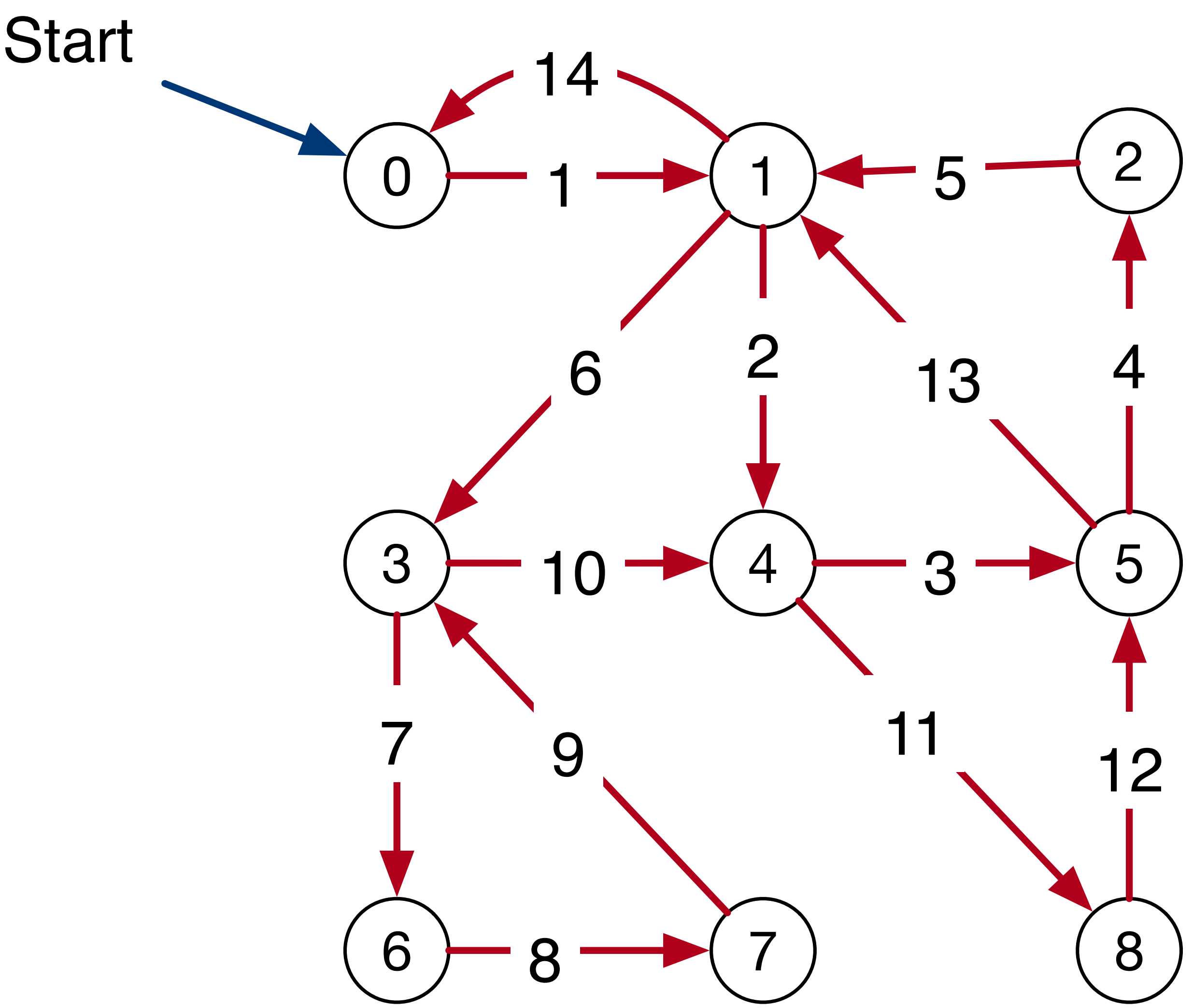
- Given a number of pairs of numbers, i.e. a directed graph and a starting location



- Find a path that starts at the specified location and uses up all edges

In Class Exercise

- A solution
  - which happens to be a cycle



In Class Exercise

- Can we construct a partial solution?

In Class Exercise

- Can we construct a partial solution?
  - Yes:
    - We start with the start location
      - We then add other locations at the end

In Class Exercise

- Can we verify if a partial solution is invalid?

In Class Exercise

- Can we verify if a partial solution is invalid?
  - Yes:
    - If there is no unused edge starting with the last node, but there are unused edges, then the solution is invalid

In Class Exercise

- Can we verify if the solution is complete?

In Class Exercise

- Can we verify if the solution is complete?
  - Yes.
    - The solution is complete if there are no more edges left

In Class Exercise

- Setting up
  - Define a data structure for edges
  - Define a data structure for the itinerary

In Class Exercise

```
edges = [ (0,1), (1,0), (1,3), (1,4), (2,1),
          (3,4), (3,6), (4,5), (4,8), (5,1),
          (5,2), (6,7), (7,3), (8,5) ]

current_itinerary = [0]
```

In Class Exercise

- Stop condition:
  -
```
if not edges:
        return current_itinerary
```

In Class Exercise

- Schema

```
current = []
def get_itinerary(edges, current):
    if not edges:
        return current_itinerary
    ## use an edge to add to the itinerary
        ## use recursion
    ## else backtrack!


    return None
```

In Class Exercise

- Expand

```
def get_itinerary(edges, current):
    print('edges, itinerary', edges, current, current[-1])
    if not edges:
        return current
    for edge in edges:
        if edge[0] == current[-1]:
            current.append(edge[1])
            current_edges = [e for e in edges if not e == edge]
            result = get_itinerary(current_edges, current)
            if result:
                return result
            current.pop()
    return None
```

In Class Exercise 2

- Gray Codes:
  - List all numbers from 0 to $2^n - 1$ so that consecutive numbers differ only in one bit in the binary representation
  - Examples:
    - [0, 1, 5, 13, 12, 8, 9, 11, 10, 2, 3, 7, 15, 14, 6, 4]
    - [0, 1, 5, 4, 12, 8, 9, 13, 15, 7, 6, 2, 3, 11, 10, 14]

In Class Exercise 2

- Calculate the Hamming weight of a number
    - Number of one-bits
        - Use binary operations

```
def hamming(a):
    count = 0
    while(a):
        if a&1 == 1:
            count += 1
        a = a>>1
    return count
```

In Class Exercise 2

```
for i in range(15):
    print('{:05b}'.format(i),
          hamming(i))
```

```
000000 0  010000 1
000001 1  010001 2
000010 1  010010 2
000011 2  010011 3
000100 1  010100 2
000101 2  010101 3
000110 2  010110 3
000111 3  010111 4
001000 1  011000 2
001001 2  011001 3
001010 2  011010 3
001011 3  011011 4
001100 2  011100 3
001101 3  011101 4
001110 3  011110 4
001111 4  011111 5
```

In Class Exercise 2

- Use backtracking
  - Can we use partial solutions?
  - Can we verify if a partial solution is invalid?
  - Can we verify if the solution is complete?

In Class Exercise 2

- Use backtracking
  - Can we use partial solutions?
    - A partial list of numbers
  - Can we verify if a partial solution is invalid?
    - Cannot find another number to add to it
  - Can we verify if the solution is complete?
    - All numbers are used up

In Class Exercise 2

```
numbers = [2, 3, 4, 5, 6, 7 ]
current = [0,1]

def gray_code(current, numbers):
    if not numbers:
        return current
    for num in numbers:
        If we can add num, do it
        recursive call, return if successful
        undo num
    return None
```

In Class Exercise 2

```
def gray_code(current, numbers):
    if not numbers:
        return current
    for num in numbers:
        if hamming(current[-1]^num) == 1:
            current_numbers = [n for n in numbers if n != num]
            current.append(num)
            result = gray_code(current, current_numbers)
            if result:
                return result
            current.pop()
    return None
```

In Class Exercise 2

```
numbers = list(range(2,16))
random.shuffle(numbers)
print(gray_code([0,1], numbers))
```