Back-Tracking

Thomas Schwarz, SJ

Complete Enumeration

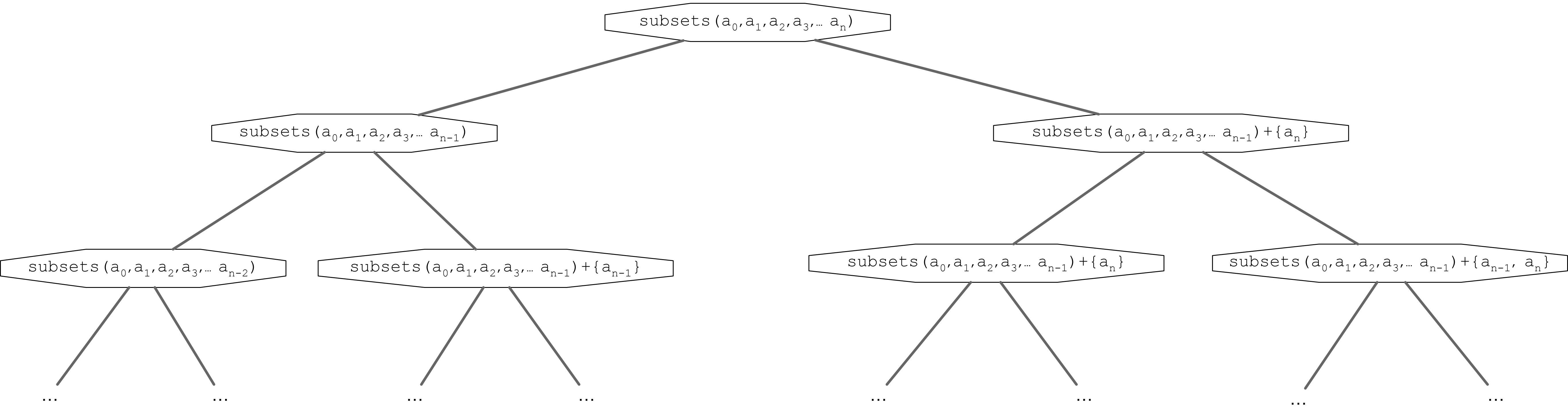
- You are given:
  - A set of numbers, e.g. $\mathbb{S} = \{1,5,12,14,19,20,21\}$
  - A target number t
- Your task is to find a subset of $\mathbb{S}$ such that the sum of the letters in the subset is as close to t as possible.

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

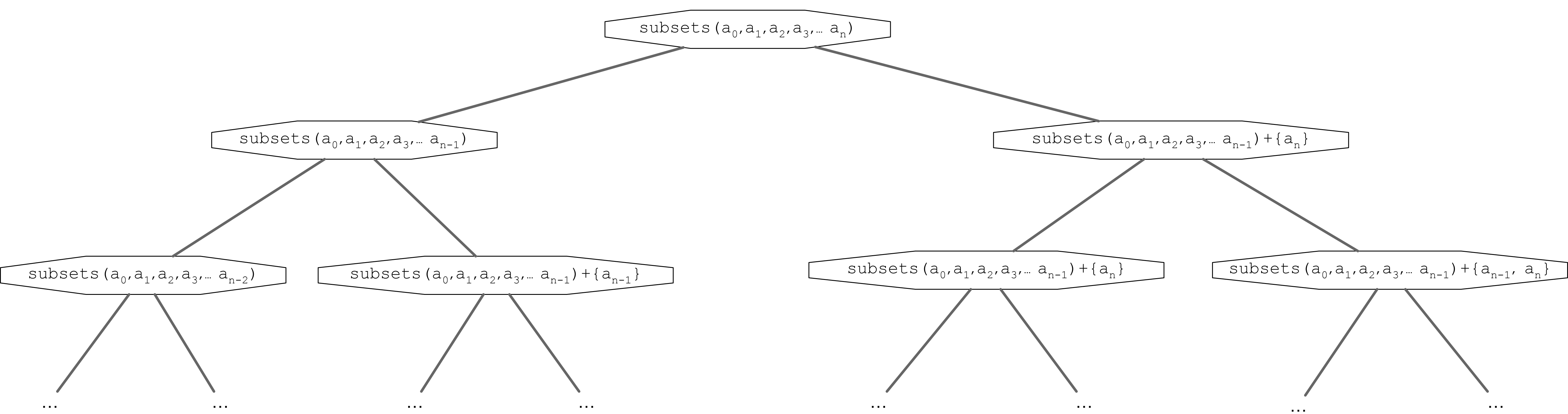
Complete Enumeration

- 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

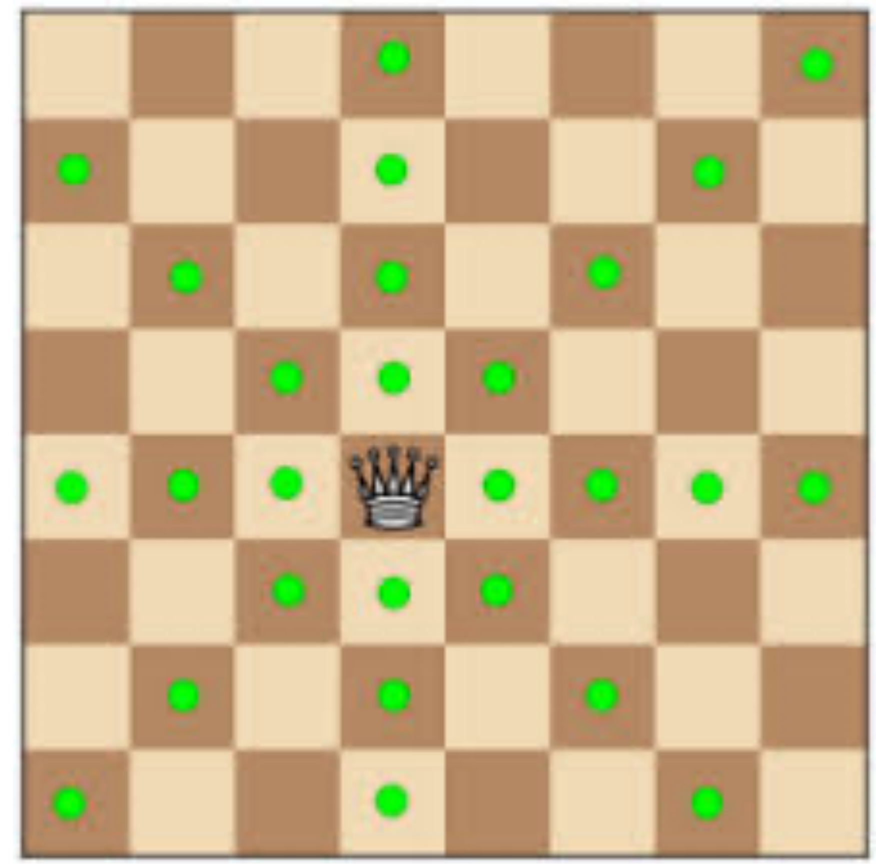
- Complete enumeration of subsets generates 2^n subsets
  - Therefore, is exponential
- In general: complete enumeration with recursion creates a call tree with b^n or b^{n+1} leaves

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

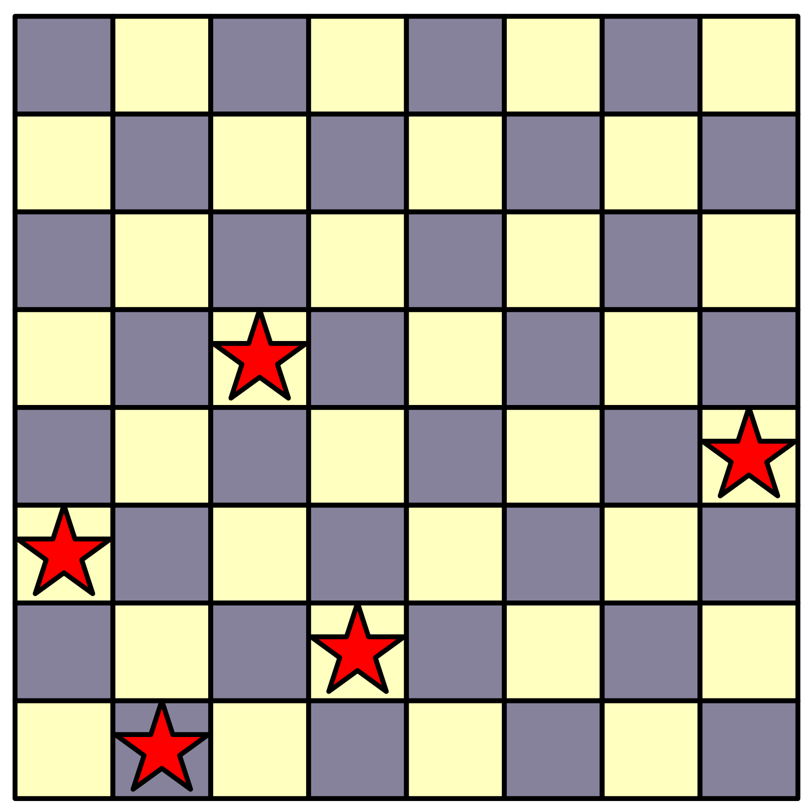
- Strategy:
    - We notice that there can be only one queen per column
    - And that there has to be one in every column to get the total number to n

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

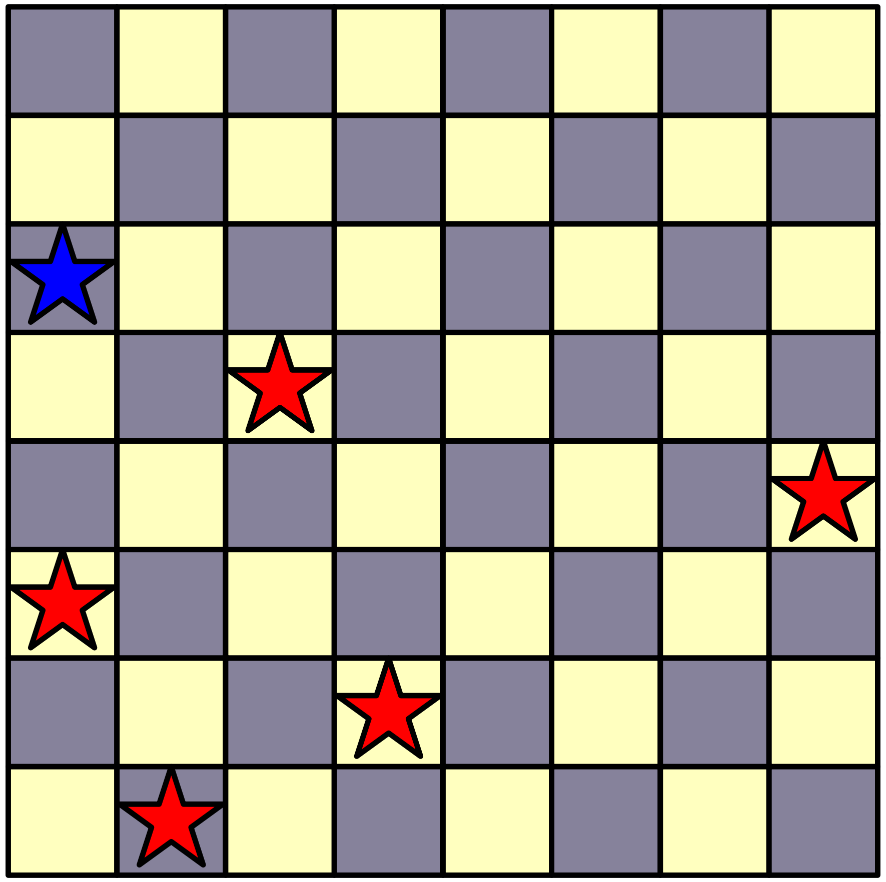
Back Tracking

- We encode the problem by having a list `board`
- i^{th} queen is located in row i and column `board[i]`
  - E.g. `board = [1,3,0,`**`7`**`,2]`



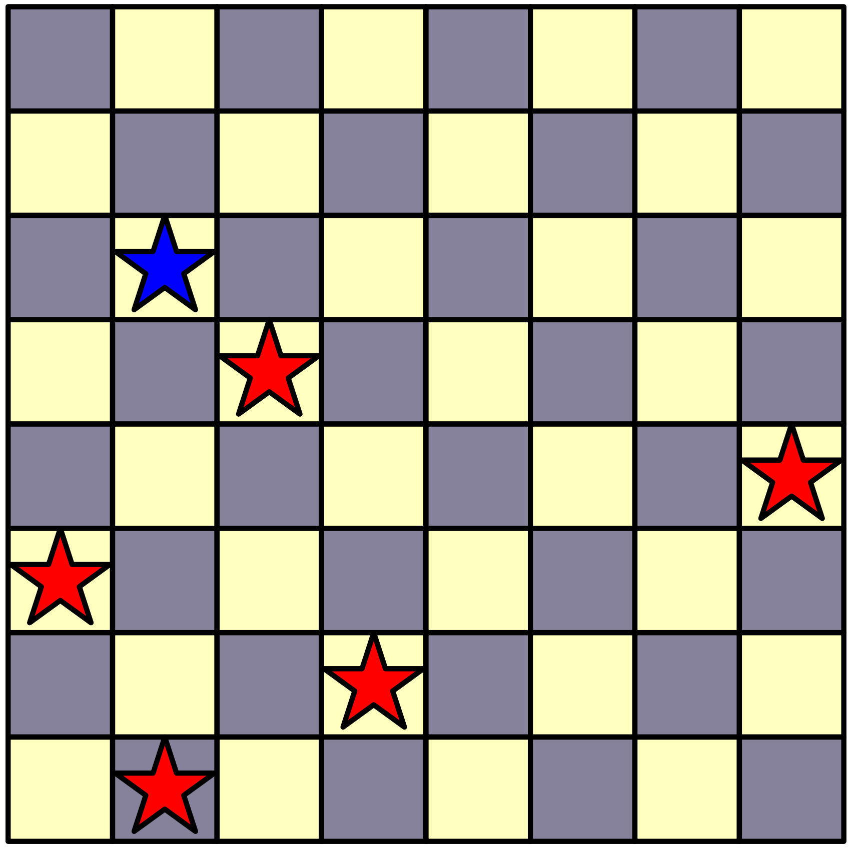
Back Tracking

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



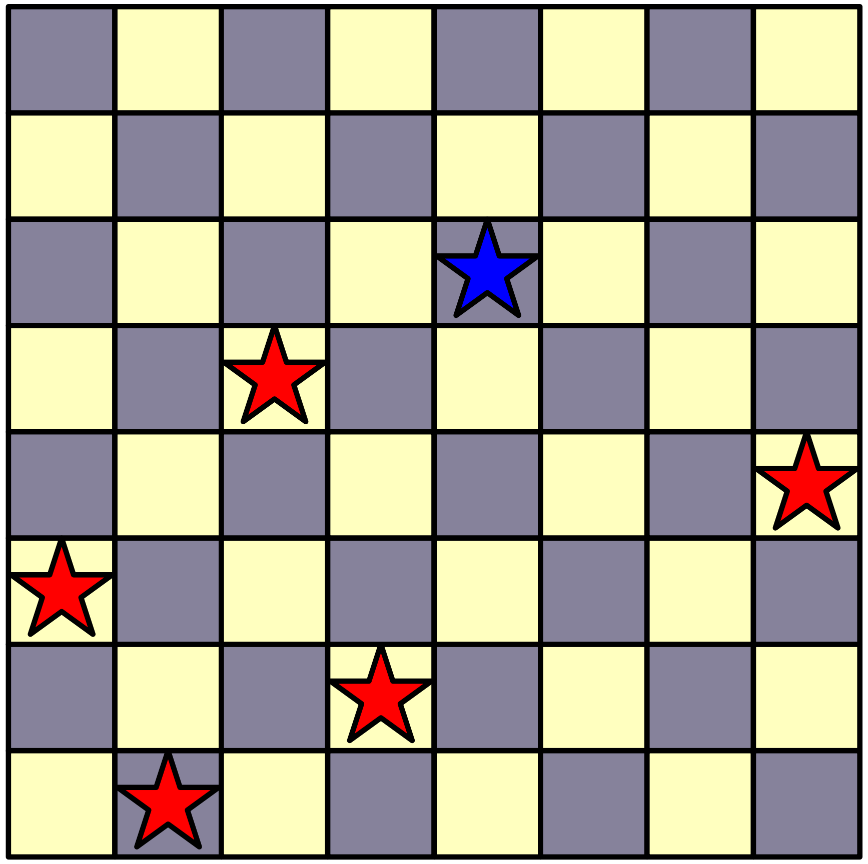
Back Tracking

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



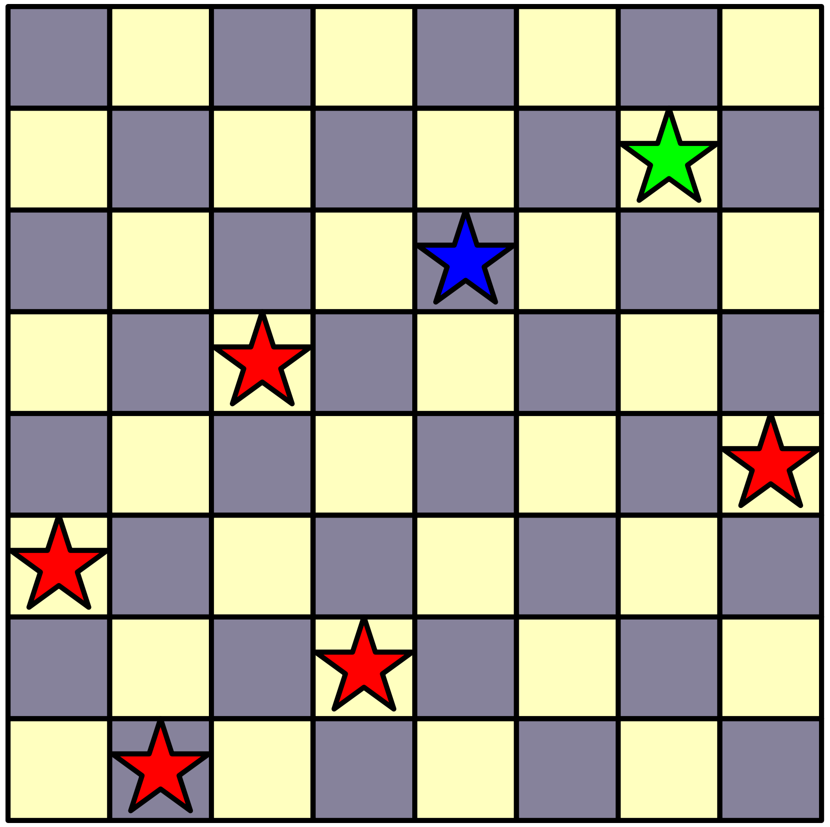
Back Tracking

- E.g. `board[1,3,0,7,2]`
  - 4 works
    - And then we try further recursively



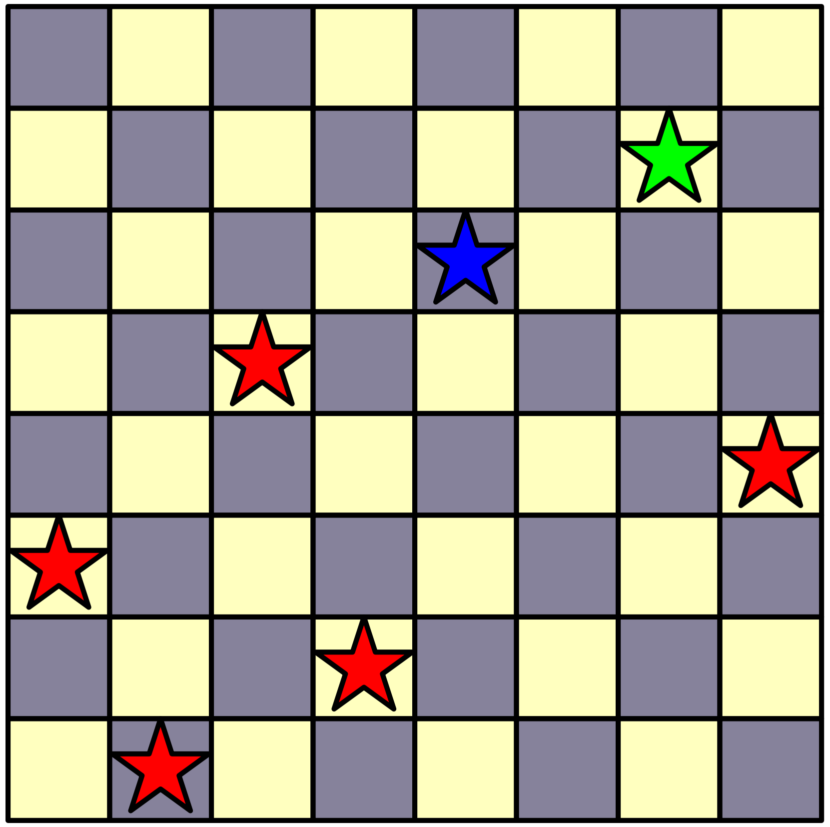
Back Tracking

- E.g. `board[1,3,0,7,2]`
    - 4 works
        - And then we try further recursively



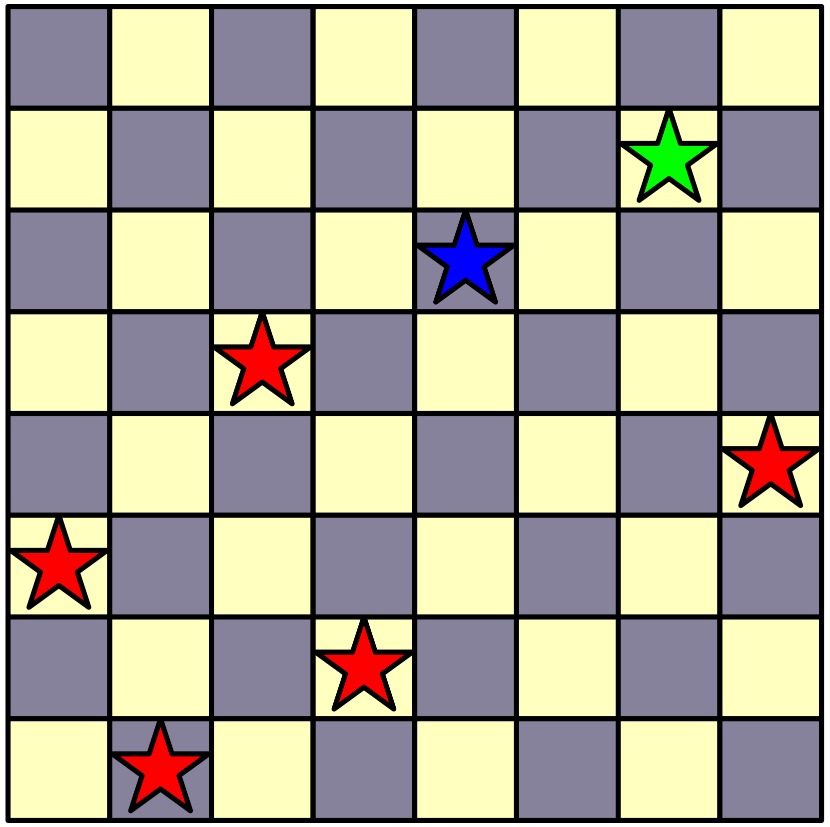
Back Tracking

- E.g. `board[1,3,0,7,2]`
  - 4 works
    - And try more, but now we are stuck



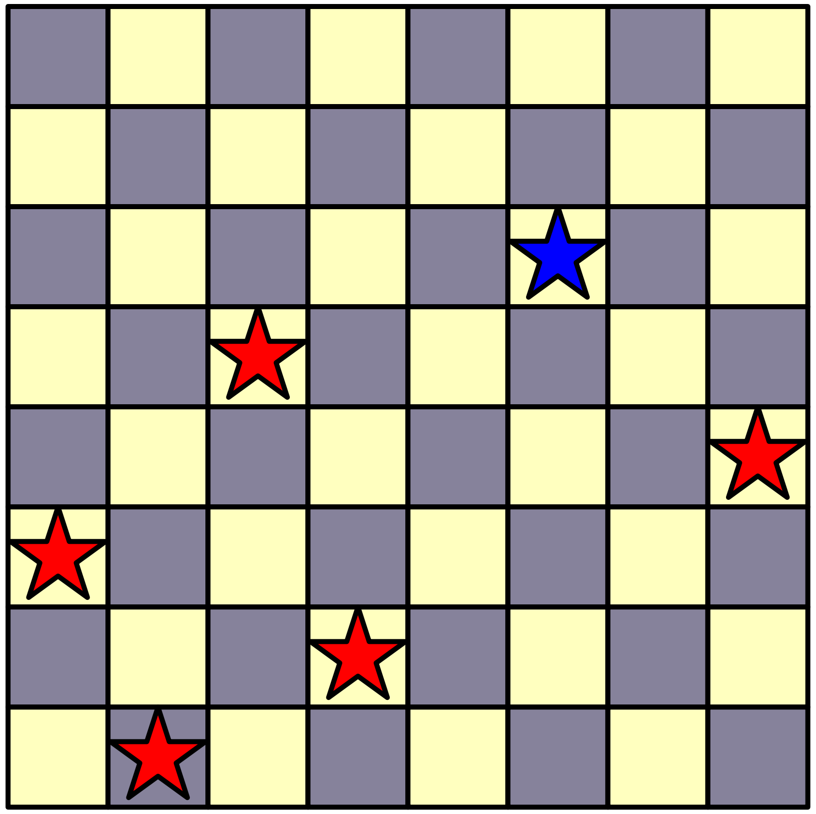
Back Tracking

- E.g. `board[1,3,0,7,2]`
  - Need to undo, but the OS-stack takes care of that
  - Our recursive calls just return



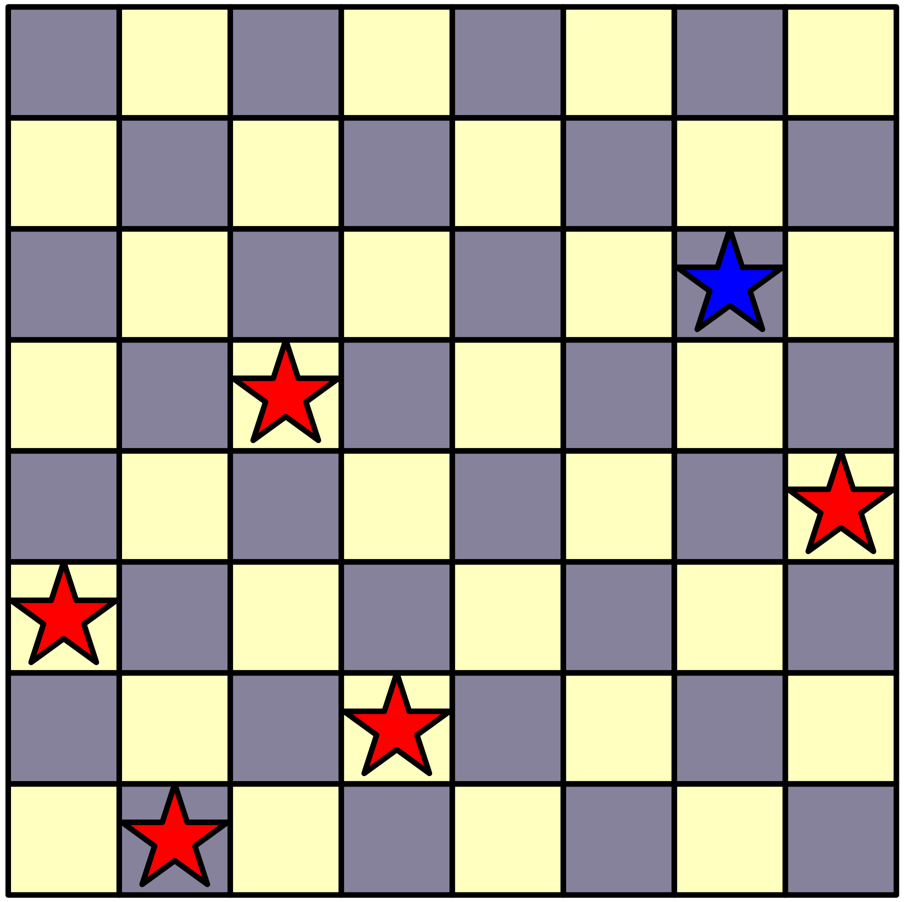
Back Tracking

- E.g. `board[1,3,0,7,2]`
  - Back in this situation and now 5 works
  - But you can already see that the next step does not



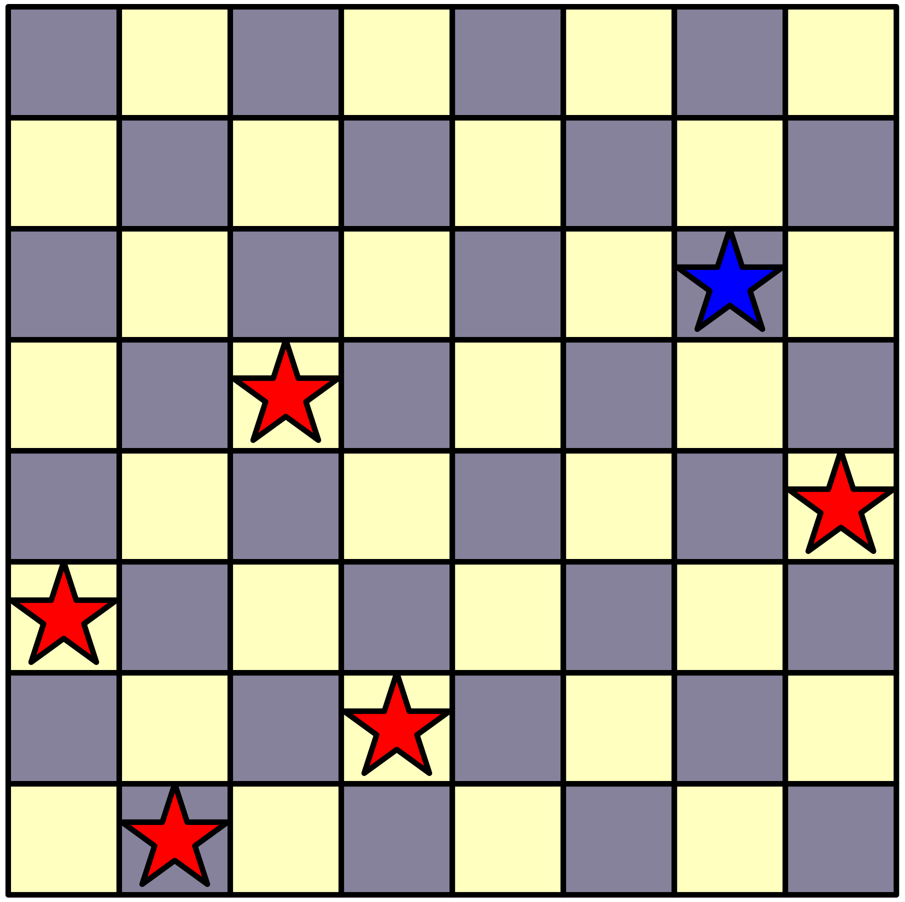
Back Tracking

- E.g. `board[1,3,0,7,2]`
    - Back in this situation and now 6 works



Back Tracking

- E.g. `board[1,3,0,7,2]`
  - Recursion tries to place something in row 6
  - Nothing works, so this was a dead-end



Back Tracking

- We implement this as a double loop
  - Inner loop tries placement
  - Outer loop is implemented via recursion

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

- We now count how many solutions there are

```
def n_queens(n, board = []):
    if n == len(board):
        return 1

    count = 0
    for col in range(n):
        board.append(col)
        if is_valid(board):
            count += n_queens(n, board)
        board.pop()
    return count
```

Back Tracking

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

```
def n_queens(n, board = []):
    if n == len(board):
        return 1

    count = 0
    for col in range(n):
        board.append(col)
        if is_valid(board):
            count += n_queens(n, board)
        board.pop()
    return count
```

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