CSE 143
Lecture 13

Recursive Backtracking

slides created by Ethan Apter
http://www.cs.washington.edu/143/

Definitions

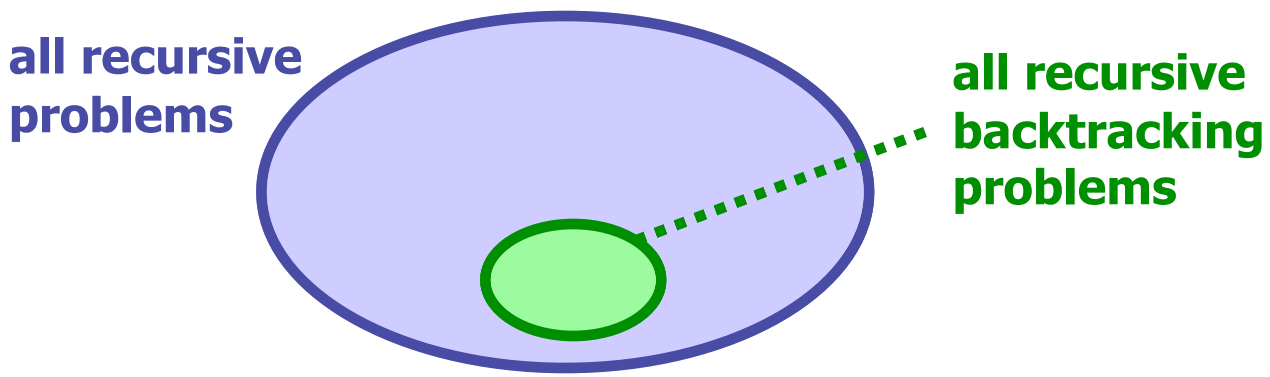
- **recursive backtracking:** backtracking using recursion
- **backtracking**: a brute-force technique for finding solutions. This technique is characterized by the the ability to undo ("backtrack") when a potential solution is found to be invalid.
- **brute-force**: not very smart, but very powerful
  - more specifically: not very efficient, but will find a valid solution (if a valid solution exists)
- Even though backtracking is a brute-force technique, it is actually a relatively efficient brute-force technique
  - it's still slow, but it's better than some approaches

Wait, what?

- Common question: what's the difference between "recursion" and "recursive backtracking"?

- recursion: any method that calls itself (recurses) to solve a problem

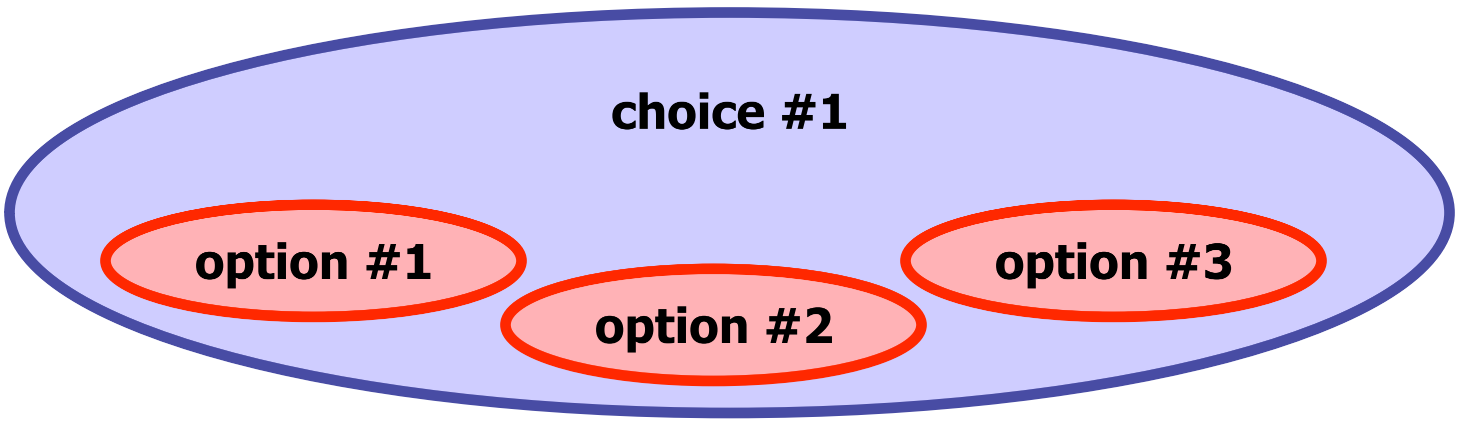
- recursive backtracking: a specific technique (backtracking) that is expressed through recursion
  - backtracking algorithms are easily expressed with recursion



Basic Idea

- We want to try every possibility to see if it's a solution
  - unless we already know it's invalid

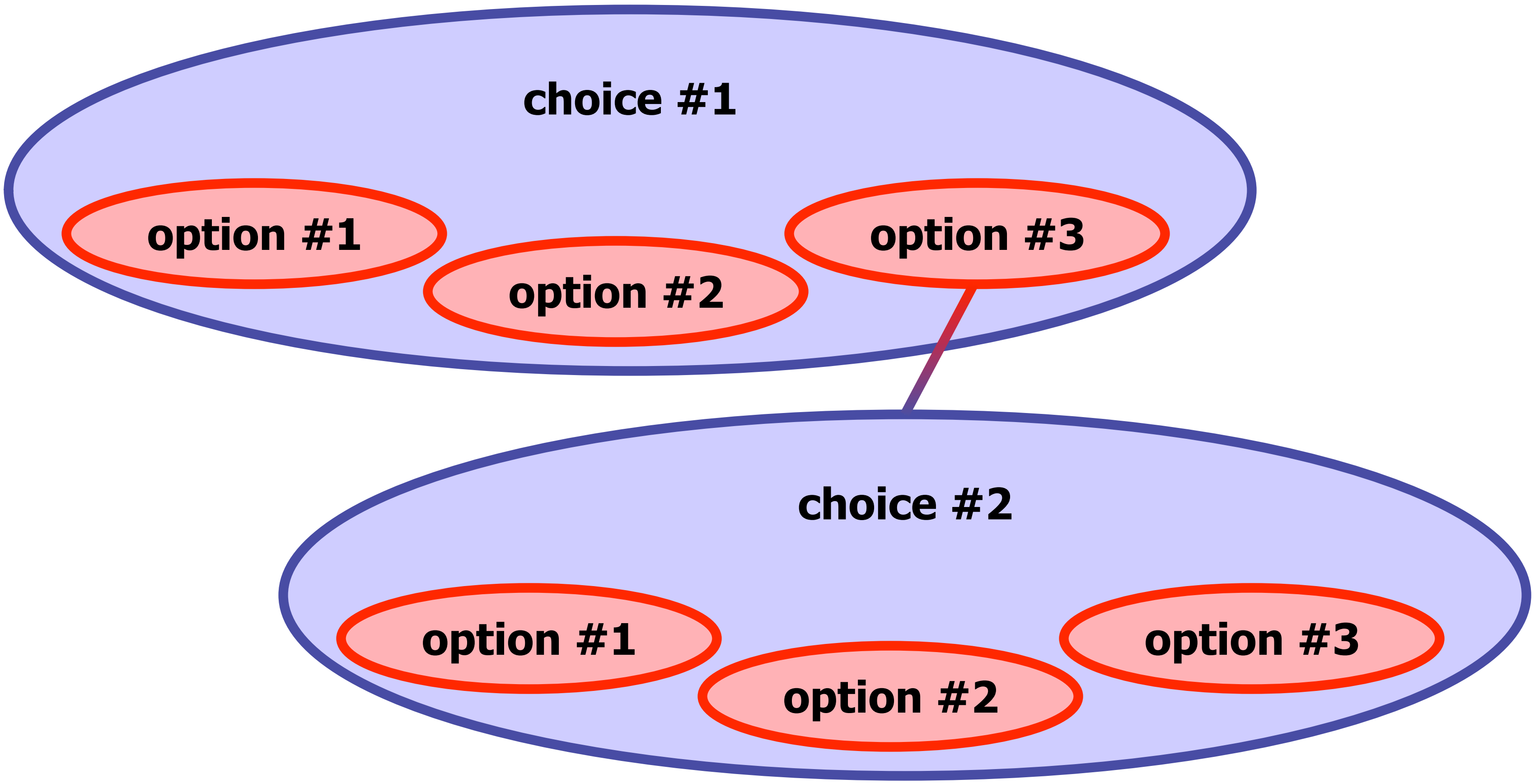
- We can view this as a sequence of choices. The first choice might look something like this:



- What happens if we select one of the options?

Basic Idea

- Suppose we choose option #3:



- We are presented with another choice (that is based on the option we chose)

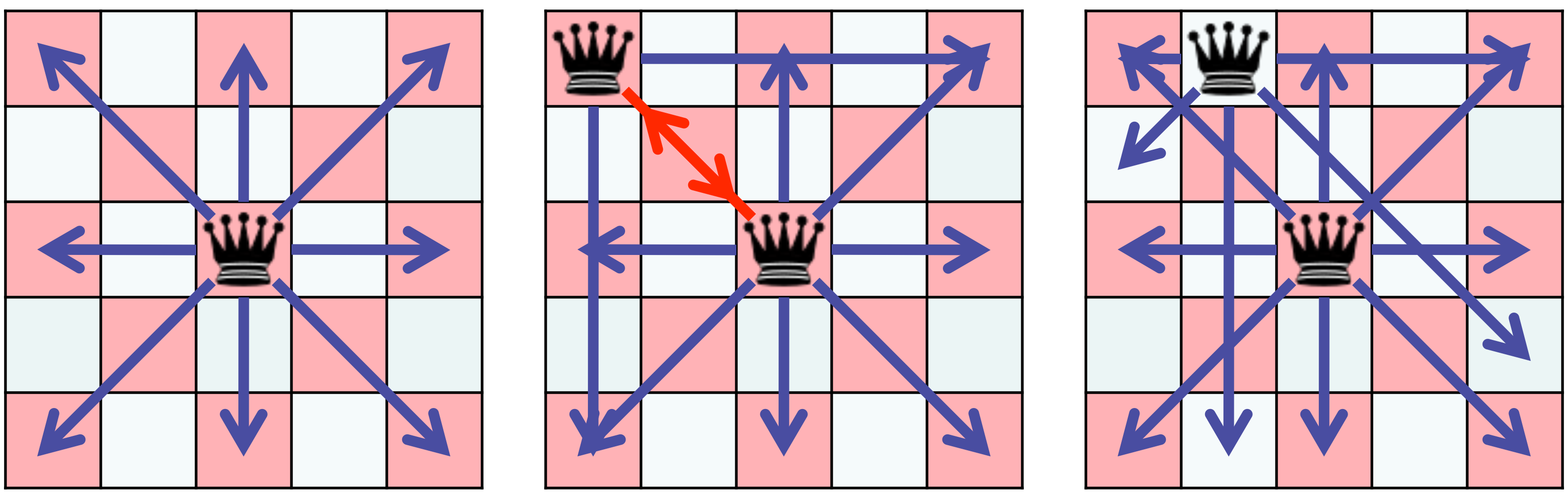
Basic Idea

- And this sequence of choices continues until:
  - you decide you've made a bad choice somewhere along the sequence and want to backtrack
  - you decide you've arrived at a perfectly valid solution
- But this process gets pretty hard to draw, because it fans out so much
  - so you'll have to use your imagination
- This is also why brute-force techniques are slow
  - exploring every possibility takes time because there are so many possibilities

8 Queens

- 8 Queens is a classic backtracking problem

- 8 Queens: place 8 queens on an 8x8 chessboard so that no queen threatens another
  - queens can move in a straight line horizontally, vertically, or diagonally any number of spaces



possible moves **threatened!** **safe!**

8 Queens

- One possible approach:
  - on an 8x8 chessboard, there are 64 locations
  - each of these locations is a potential location to place the first queen (this is a choice!)
  - after we place the first queen, there are 63 remaining locations to place the second queen
    - clearly, some of these won't work, because the second queen will threaten the first queen.
  - after we place the second queen, there are 62 remaining locations to place the third queen
  - and so on

- So, there are 178,462,987,637,760 possibilities!
  - 178,462,987,637,760 = 64*63*62*61*60*59*58*57

8 Queens

- That’s a lot of choices!
- Remember that we’re using a brute-force technique, so we have to explore all possible choices
  - now you can really see why brute-force techniques are slow!
- However, if we can refine our approach to make fewer choices, we can go faster
  - we want to be clever about our choices and make as few choices as possible
- Fortunately we can do a lot better

8 Queens

- Key observation:
  - all valid solutions to 8 Queens will have exactly 1 queen in each row and exactly 1 queen in each column (otherwise the queens *must* threaten each other)
- There are exactly 8 queens, 8 rows, and 8 columns
- So rather than exploring 1-queen-per-board-location, we can explore 1-queen-per-row or 1-queen-per-column
  - it doesn't matter which
- We'll explore 1-queen-per-column

8 Queens

- When exploring column-by-column, we have to decide which row to place the queen for a particular column

- There are 8 columns and 8 rows
  - so we've reduced our possible choices to $8^8 = 16{,}777{,}216$

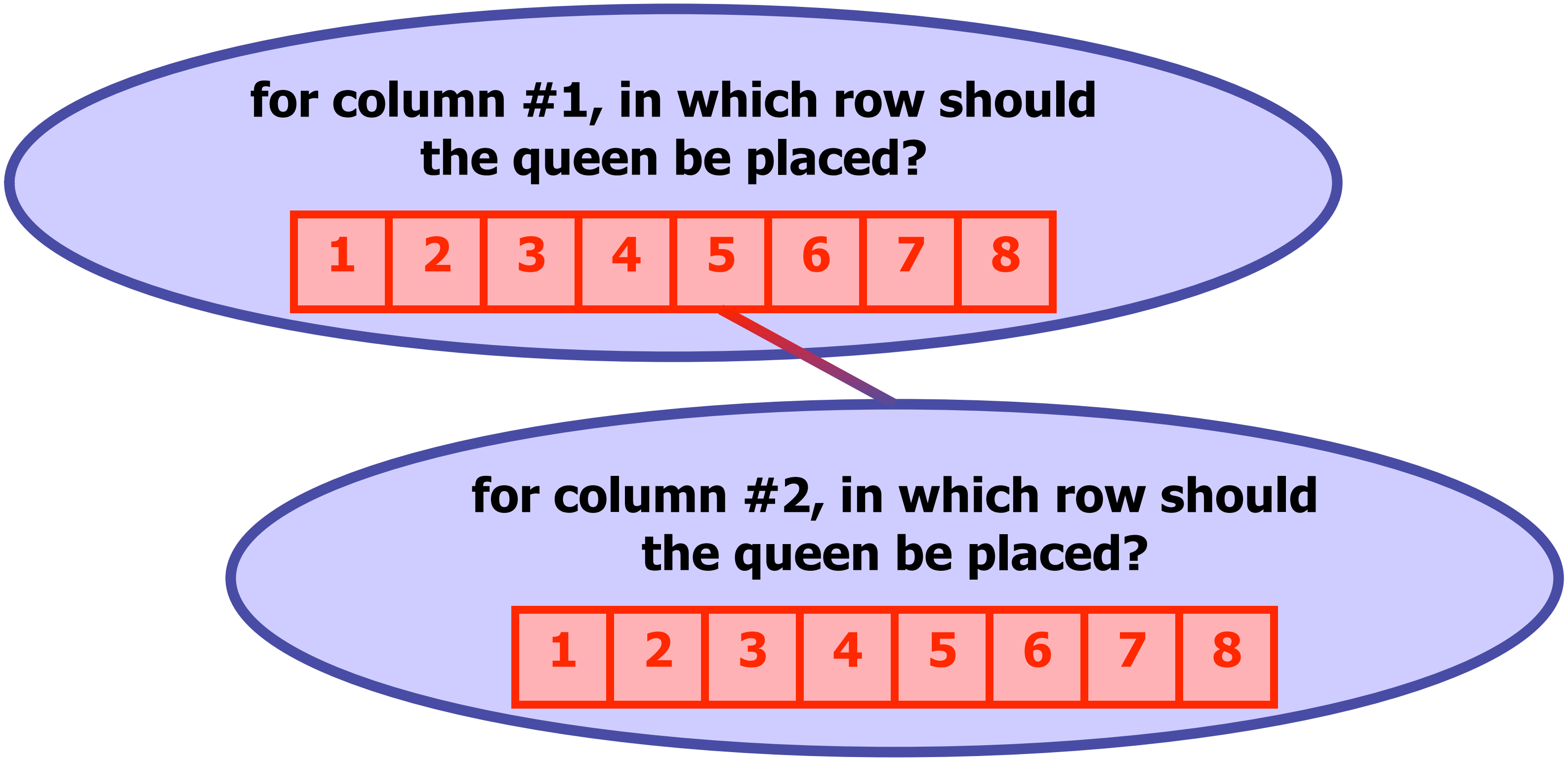
- So our first decision looks something like this:

for column #1, in which row should the queen be placed?

1	2	3	4	5	6	7	8

8 Queens

- If we choose to place the queen in row #5, our decision tree will look like this:



- Keep in mind that our second choice (column #2) is affected by our first choice (column #1)

8 Queens

- So, why are we using backtracking?
  - because backtracking allows us to undo previous choices if they turn out to be mistakes
- Notice that as we choose which row to place each queen, we don't actually know if our choices will lead to a solution
  - we might be wrong!
- If we are wrong, we want to undo the minimum number of choices necessary to get back on the right track
  - this also saves as much previous work as possible

8 Queens

- It's clear that we could explore the possible choices for the first column with a loop:

```
for (int row = 1; row <= 8; row++) // explore column 1
```

- So we could solve the whole problem with nested loops somewhat like this:

```
for (int row = 1; row <= 8; row++) // column 1
  for (int row = 1; row <= 8; row++) // column 2
    for (int row = 1; row <= 8; row++) // column 3
      ...
        for (int row = 1; row <= 8; row++) // column 8
```

- But we can write this more elegantly with recursion

8 Queens

- Recursive backtracking problems have somewhat of a general form
- This form is easier to see (and the code is easier to understand) if we remove some of the low-level details from our recursive backtracking code
- To do this, we'll be using some code Stuart Reges wrote. Stuart's code is based off code written by one of his former colleagues named Steve Fisher
- We're going to use this code to solve N Queens
  - just like 8 queens, but now we can have N queens on an NxN board (where N is any positive number)

N Queens

- What low-level methods do we need for N Queens?
  - We need a constructor that takes a size (to specify N):

    ```
    public Board(int size)
    ```

  - We need to know if it’s safe to place a queen at a location

    ```
    public boolean safe(int row, int col)
    ```

  - We need to be able to place a queen on the board

    ```
    public void place(int row, int col)
    ```

  - We need to be able to remove a queen from the board, because we might make mistakes and need to backtrack

    ```
    public void remove(int row, int col)
    ```

  - And we need some general information about the board

    ```
    public void print()
    public int size()
    ```

N Queens

- Assume we have all the previous code
- With that taken care of, we just have to find a solution!
  - easy, right?
- Let's write a method called solve to do this:

```
public static void solve(Board b) {
    ...
}
```

- Unfortunately, **`solve`** doesn't have enough parameters for us to do our recursion
  - so let's make a private helper method

N Queens

- Our private helper method:

```
private static boolean explore(...) {
    ...
}
```

- What parameters does explore need?
  - it needs a Board to place queens on
  - it needs a column to explore
    - this is a little tricky to see, but this will let each method invocation work on a different column

- Updated helper method:

```
private static boolean explore(Board b, int col) {
    ...
}
```

N Queens

- Well, now what?

- We don't want to waste our time exploring dead ends
  - so, if someone wants us to explore column #4, we should require that columns #1, #2, and #3 all have queens and that these three queens don't threaten each other
  - we'll make this a precondition (it's a *private* method, after all)

- So, now our helper method has a precondition:

```
// pre : queens have been safely placed in previous
//       columns
```

N Queens

- Time to write our method
- We know it's going to be recursive, so we need at least:
  - a base case
  - a recursive case
- Let's think about the base case first
- What column would be nice to get to? When are we done?
  - For 8 Queens, column 9 (queens 1 to 8 placed safely)
    - column 8 is almost correct, but remember that if we're asked to explore column 8, the 8th queen hasn't yet been placed
  - For N Queens, column N+1 (queens 1 to N placed safely)

N Queens

- This is our base case!

- Let’s update our helper code to include it:

```
private static boolean explore(Board b, int col) {
    if (col > b.size()) {
        return true;
    } else {
        ...
    }
}
```

- Well, that was easy

- What about our recursive case?

N Queens

- For our recursive case, suppose we've already placed queens in previous columns
- We want to try placing a queen in all possible rows for the current column
- We can try all possible rows using a simple for loop:

```
for (int row = 1; row <= board.size(); row++) {
    ...
}
```

- This is the same for loop from before!
  - remember, even though we're using recursion, we still want to use loops when appropriate

N Queens

- When do we want to try placing a queen at a row for the specified column?
  - only when it is safe to do so!
  - otherwise this location is already threatened and won’t lead us to a solution

- We can update our code:

```
for (int row = 1; row <= board.size(); row++) {
    if (b.safe(row, col)) {
        ...
    }
}
```

- We’ve picked our location and determined that it’s safe
  - now what?

N Queens

- We need to place a queen at this spot and decide if we can reach a solution from here
  - if only we had a method that would explore a Board from the next column and decide if there's a solution...
  - oh wait! That's what we're writing
- We can update our code to place a queen and recurse:

```
for (int row = 1; row <= board.size(); row++) {
    if (b.safe(row, col)) {
        b.place(row, col);
        explore(b, col + 1);
        ...
    }
}
```

You might be tempted to write `col++` here instead, but that won't work. Why not?

N Queens

- Also, we don't want to call **`explore`** quite like that
  - **`explore`** returns a **`boolean`**, telling us whether or not we succeeded in finding a solution (true if found, false otherwise)

- What should we do if **`explore`** returns true?
  - stop exploring and return true (a solution has been found)

- What should we do if **`explore`** returns false?
  - well, the queens we've placed so far don't lead to a solution
  - so, we should remove the queen we placed most recently and try putting it somewhere else

N Queens

- Updated code:

```
for (int row = 1; row <= board.size(); row++) {
    if (b.safe(row, col)) {
        b.place(row, col);
        if (explore(b, col + 1)) {
            return true;
        }
        b.remove(row, col);
    }
}
```

This pattern (make a choice, recurse, undo the choice) is *really* common in recursive backtracking

- We're almost done. What should we do if we've tried placing a queen at every row for this column, and no location leads to a solution?
  - No solution exists, so we should return false

N Queens

- And we're done! Here's the final code for explore:

```
private static boolean explore(Board b, int col) {
    if (col > b.size()) {
        return true;
    } else {
        for (int row = 1; row <= board.size(); row++) {
            if (b.safe(row, col)) {
                b.place(row, col);
                if (explore(b, col + 1)) {
                    return true;
                }
                b.remove(row, col);
            }
        }
        return false;
    }
}
```

N Queens

- Well, actually we still need to write **`solve`**
  - don't worry, it's easy!

- We'll have **`solve`** print out the solution if **`explore`** finds one. Otherwise, we'll have it tell us there's no solution

- Code for **`solve`**:

```
public static void solve(Board b) {
    if (explore(b, 1)) {
        System.out.println("One solution is as follows:");
        b.print();
    } else {
        System.out.println("No solution");
    }
}
```

N Queens

- We're really done and everything works
  - try running the code yourself!
  - I think it's pretty cool that such succinct code can do so much

- There's also an animated version of the code
  - it shows the backtracking process in great detail
  - if you missed lecture (or if you just want to see the animation again), download **`queens.zip`** from the class website and run **`Queens2.java`**