Review: Recursive Problem-Solving

- When we use recursion, we solve a problem by reducing it to a simpler problem of the same kind.

- We keep doing this until we reach a problem that is simple enough to be solved directly.

- This simplest problem is known as the base case.

```java
def printSeries(int n1, int n2) {
if (n1 == n2) { // base case
    System.out.println(n2);
} else {
    System.out.println(n1 + " , ");
    printSeries(n1 + 1, n2);
}
}
```

- The base case stops the recursion, because it doesn't make another call to the method.
Review: Recursive Problem-Solving (cont.)

- If the base case hasn't been reached, we execute the recursive case.
  
  ```java
  public static void printSeries(int n1, int n2) {
      if (n1 == n2) { // base case
          System.out.println(n2);
      } else { // recursive case
          System.out.print(n1 + "", "");
          printSeries(n1 + 1, n2);
      }
  }
  ```

- The recursive case:
  - reduces the overall problem to one or more simpler problems of the same kind
  - makes recursive calls to solve the simpler problems

Raising a Number to a Power

- We want to write a recursive method to compute
  \[ x^n = x \times x \times \ldots \times x \]
  where \( x \) and \( n \) are both integers and \( n \geq 0 \).

- Examples:
  - \( 2^{10} = 2 \times 2 \times 2 \times 2 \times 2 \times 2 = 1024 \)
  - \( 10^5 = 10 \times 10 \times 10 \times 10 \times 10 = 100000 \)

- Computing a power recursively:
  \[
  2^{10} = 2^{2^9} = 2^{\left(2^2 \right)^2} = \cdots
  \]

- Recursive definition:
  \[
  x^n = x \times x^{n-1} \quad \text{when} \quad n > 0
  
  x^0 = 1
  \]
Power Method: First Try

```java
class Power {
    public static int power1(int x, int n) {
        if (n < 0)
            throw new IllegalArgumentException("n must be >= 0");
        if (n == 0)
            return 1;
        else
            return x * power1(x, n-1);
    }
}
```

Example: `power1(5, 3)`

<table>
<thead>
<tr>
<th>x</th>
<th>n</th>
<th>5</th>
<th>n+1</th>
<th>5</th>
<th>n+2</th>
<th>5</th>
<th>n+3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Power Method: Second Try

- There’s a better way to break these problems into subproblems. For example: 2^10 = (2*2*2*2*2)*(2*2*2*2*2) = (2^5) * (2^5) = (2^5)^2
- A more efficient recursive definition of \(x^n\) (when \(n > 0\)):
  \[x^n = (x^{n/2})^2\] when \(n\) is even
  \[x^n = x * (x^{n/2})^2\] when \(n\) is odd (using integer division for \(n/2\))
- Let’s write the corresponding method together:

```java
class Power {
    public static int power2(int x, int n) {
    }
}
```
Analyzing power2

- How many method calls would it take to compute $2^{1000}$?

```java
power2(2, 1000)
power2(2, 500)
power2(2, 250)
power2(2, 125)
power2(2, 62)
power2(2, 31)
power2(2, 15)
power2(2, 7)
power2(2, 3)
power2(2, 1)
power2(2, 0)
```

- Much more efficient than power1() for large $n$.
- It can be shown that it takes approx. $\log_2 n$ method calls.

An Inefficient Version of power2

- What's wrong with the following version of power2()?

```java
public static int power2Bad(int x, int n) {
    // code to handle n < 0 goes here...
    if (n == 0)
        return 1;
    if ((n % 2) == 0)
        return power2(x, n/2) * power2(x, n/2);
    else
        return x * power2(x, n/2) * power2(x, n/2);
}
```
Review: Processing a String Recursively

- A string is a recursive data structure. It is either:
  - empty ("")
  - a single character, followed by a string

- Thus, we can easily use recursion to process a string.
  - process one or two of the characters
  - make a recursive call to process the rest of the string

- Example: print a string vertically, one character per line:
  ```java
  public static void printVertical(String str) {
      if (str == null || str.equals("")) {
          return;
      }
      System.out.println(str.charAt(0)); // first char
      printVertical(str.substring(1));   // rest of string
  }
  ```

Removing Vowels from a String

- Let’s design a recursive method called `removeVowels()`.

- `removeVowels(str)` should return a string in which all of the vowels in the string `str` have been removed.

- Example:
  ```java
  removeVowels("recurse")
  ```
  should return
  ```java
  "rcrs"
  ```

- Thinking recursively:
Removing Vowels from a String (cont.)

- Put the method definition here:

Recursive Backtracking: the n-Queens Problem

- Find all possible ways of placing n queens on an n x n chessboard so that no two queens occupy the same row, column, or diagonal.

- Sample solution for n = 8:

![N-Queens Solution Diagram]

- This is a classic example of a problem that can be solved using a technique called recursive backtracking.
Recursive Strategy for n-Queens

- Consider one row at a time. Within the row, consider one column at a time, looking for a “safe” column to place a queen.
- If we find one, place the queen, and make a recursive call to place a queen on the next row.
- If we can’t find one, backtrack by returning from the recursive call, and try to find another safe column in the previous row.

Example for $n = 4$:
- row 0:
  
<table>
<thead>
<tr>
<th>col 0: safe</th>
</tr>
</thead>
<tbody>
<tr>
<td>q</td>
</tr>
</tbody>
</table>

- row 1:
  
  | col 0: same col |
  | col 1: same diag |
  | col 2: safe     |
  | q               |

4-Queens Example (cont.)

- row 2:
  
  | col 0: same col |
  | col 1: same diag |
  | col 2: same col/diag |
  | col 3: same diag |
  | q               |

- We’ve run out of columns in row 2!
- Backtrack to row 1 by returning from the recursive call.
  - pick up where we left off
  - we had already tried columns 0-2, so now we try column 3:
    
    | we left off in col 2 |
    | try col 3: safe |
    | q               |

- Continue the recursion as before.
4-Queens Example (cont.)

- row 2:

- row 3:

- Backtrack to row 2:

- Backtrack to row 1. No columns left, so backtrack to row 0!

4-Queens Example (cont.)

- row 0:

- row 1:

- row 2:

- row 3: A solution!
findSafeColumn() Method

```java
public void findSafeColumn(int row) {
    if (row == boardSize) {  // base case: a solution!
        solutionsFound++;
        displayBoard();
        if (solutionsFound >= solutionTarget)
            System.exit(0);
        return;
    }
    for (int col = 0; col < boardSize; col++) {
        if (isSafe(row, col)) {
            placeQueen(row, col);
            // Move onto the next row.
            findSafeColumn(row + 1);
            // If we get here, we've backtracked.
            removeQueen(row, col);
        }
    }
}
```

Note: neither row++ nor ++row will work here.

Tracing findSafeColumn()

```java
public void findSafeColumn(int row) {
    if (row == boardSize) {
        // code to process a solution goes here...
    }
    for (int col = 0; col < BOARD_SIZE; col++) {
        if (isSafe(row, col)) {
            placeQueen(row, col);
            findSafeColumn(row + 1);
            removeQueen(row, col);
        }
    }
}
```

We can pick up where we left off, because the value of col is stored in the stack frame.

backtrack!

backtrack!
Template for Recursive Backtracking

```java
void findSolutions(n, other params) {
  if (found a solution) {
    solutionsFound++;
    displaySolution();
    if (solutionsFound >= solutionTarget)
      System.exit(0);
    return;
  }

  for (val = first to last) {
    if (isValid(val, n)) {
      applyValue(val, n);
      findSolutions(n + 1, other params);
      removeValue(val, n);
    }
  }
}
```

Template for Finding a Single Solution

```java
boolean findSolutions(n, other params) {
  if (found a solution) {
    displaySolution();
    return true;
  }

  for (val = first to last) {
    if (isValid(val, n)) {
      applyValue(val, n);
      if (findSolutions(n + 1, other params))
        return true;
      removeValue(val, n);
    }
  }

  return false;
}
```
Data Structures for n-Queens

• Three key operations:
  • `isSafe(row, col)`: check to see if a position is safe
  • `placeQueen(row, col)`
  • `removeQueen(row, col)`

• A two-dim. array of booleans would be sufficient:
  ```java
  public class Queens {
      private boolean[][] queenOnSquare;
  }
  ```

• Advantage: easy to place or remove a queen:
  ```java
  public void placeQueen(int row, int col) {
      queenOnSquare[row][col] = true;
  }
  public void removeQueen(int row, int col) {
      queenOnSquare[row][col] = false;
  }
  ```

• Problem: `isSafe()` takes a lot of steps. What matters more?

---

Additional Data Structures for n-Queens

• To facilitate `isSafe()`, add three arrays of booleans:
  ```java
  private boolean[] colEmpty;
  private boolean[] upDiagEmpty;
  private boolean[] downDiagEmpty;
  ```

• An entry in one of these arrays is:
  - `true` if there are no queens in the column or diagonal
  - `false` otherwise

• Numbering diagonals to get the indices into the arrays:
  - `upDiag = row + col`
  - `downDiag = (boardSize – 1) + row – col`
Using the Additional Arrays

- Placing and removing a queen now involve updating four arrays instead of just one. For example:

  ```java
  public void placeQueen(int row, int col) {
    queenOnSquare[row][col] = true;
    colEmpty[col] = false;
    upDiagEmpty[row + col] = false;
    downDiagEmpty[(boardSize - 1) + row - col] = false;
  }
  ```

- However, checking if a square is safe is now more efficient:

  ```java
  public boolean isSafe(int row, int col) {
    return (colEmpty[col] && upDiagEmpty[row + col] && downDiagEmpty[(boardSize - 1) + row - col]);
  }
  ```

Recursive Backtracking II: Map Coloring

- Using just four colors (e.g., red, orange, green, and blue), we want color a map so that no two bordering states or countries have the same color.

- Sample map (numbers show alphabetical order in full list of state names):

  ![Sample map](image)

  This is another example of a problem that can be solved using recursive backtracking.
Applying the Template to Map Coloring

```java
boolean findSolutions(n, other params) {
    if (found a solution) {
        displaySolution();
        return true;
    }
    for (val = first to last) {
        if (isValid(val, n)) {
            applyValue(val, n);
            if (findSolutions(n + 1, other params))
                return true;
            removeValue(val, n);
        }
    }
    return false;
}
```

<table>
<thead>
<tr>
<th>template element</th>
<th>meaning in map coloring</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>found a solution</td>
</tr>
<tr>
<td>val</td>
<td></td>
</tr>
<tr>
<td>isValid(val, n)</td>
<td></td>
</tr>
<tr>
<td>applyValue(val, n)</td>
<td></td>
</tr>
<tr>
<td>removeValue(val, n)</td>
<td></td>
</tr>
</tbody>
</table>

Consider the states in alphabetical order. colors = { red, yellow, green, blue }. 

Map Coloring Example

We color Colorado through Utah without a problem.
- Colorado:
- Idaho:
- Kansas:
- Montana:
- Nebraska:
- North Dakota:
- South Dakota:
- Utah:

No color works for Wyoming, so we backtrack...
Map Coloring Example (cont.)

Now we can complete the coloring:

Recursive Backtracking in General

- Useful for *constraint satisfaction problems* that involve assigning values to variables according to a set of constraints.
  - n-Queens:
    - variables = Queen’s position in each row
    - constraints = no two queens in same row, column, diagonal
  - map coloring
    - variables = each state’s color
    - constraints = no two bordering states with the same color
  - many others: factory scheduling, room scheduling, etc.
- Backtracking reduces the # of possible value assignments that we consider, because it never considers invalid assignments....
- Using recursion allows us to easily handle an arbitrary number of variables.
  - stores the state of each variable in a separate stack frame