

**15-0: Recursion**

- What is a really easy (small!) version of the problem, that I could solve immediately? (Base case)
- How can I make the problem smaller?
- Assuming that I could magically solve the smaller problem, how could I use that solution to solve the original problem (Recursive Case)

**15-1: Recursion – Zero Digits**

- Write a method that returns the number of digits in a number == 0

```
int numZeroDigits(int n)
```

- numZeroDigits(20201) == 2
- numZeroDigits(34) == 0
- numZeroDigits(0) == 1
- numZeroDigits(3050060) == 4

- What is the base case?
- How can we make the problem smaller?
- How can we use the solution to the smaller problem to solve the original problem?

**15-2: Recursion – Zero Digits**

```
int numZeroDigits(int n)
{
    if (n < 10)
    {
        if (n == 0)
            return 1;
        else
            return 0;
    }
    if (n % 10 == 0)
        return 1 + numZeroDigits(n / 10);
    else
        return numZeroDigits(n / 10);
}
```

**What if  $n$  is negative? 15-3: Recursion – Zero Digits**

```
int numZeroDigits(int n)
{
    if (n < 0)
        return numZeroDigits(0 - n);
    if (n < 10)
    {
        if (n == 0)
            return 1;
        else
            return 0;
    }
    if (n % 10 == 0)
        return 1 + numZeroDigits(n / 10);
    else
        return numZeroDigits(n / 10);
}
```

**What if  $n$  is negative? 15-4: Recursion – Change Pi**

- Write a method that takes as input a string, and returns a string where all instances of the substring “pi” have been replaced with “3.14”

```
changePi("pine") ==> "3.14ne"
changePi("ppippi") ==> "p3.14p3.14"
changePi("nop ie") ==> "p3.14p3.14"
```

### 15-5: Recursion – Change Pi

```
public String changePi(String str)
{
    if ((str.length() < 2)
    {
        return str;
    }
    else if (str.substring(0,2).equals("pi"))
    {
        return "3.14" + changePi(str.substring(2));
    }
    else
    {
        return str.charAt(0) + changePi(str.substring(1));
    }
}
```

### 15-6: Backtracking

- Recursion is also useful to do a “brute force” search
- Try all possibilities
- Example: 8 Queens
  - Place a queen in the first location
  - Recursively try to place the rest of the queens
  - If you succeed, great – print or return the solution
  - If you don’t succeed place the queen in the second location

### 15-7: Backtracking - 8 Queens

- 8 Queens on whiteboard

### 15-8: Backtracking - 8 Queens

- Function takes as input a partial solution
  - Base case: Partial solution is a final solution, print it out
  - Recursive Case: For each possible valid move you could make from here:
    - Make the move
    - Recursively solve the rest of the problem
      - If the recursive call succeeds, you are done
      - If the recursive call fails, undo the move, try next one

### 15-9: Backtracking - 8 Queens

- `boolean queens(int board[], int nextColumnToTry);`
  - `board` is an array of integers: `board[i]` stores the row of the queen in column `i`
  - `nextColumnToTry` is the column that we are going to try next (that is, `board[i]` stores the location of queens for  $0 \leq i < \text{nextColumnToTry}$ )

### 15-10: Backtracking - 8 Queens

- `boolean queens(int board[], int nextColumnToTry);`
  - Base Case: The problem has already been solved
  - When is the problem solved?

#### 15-11: Backtracking - 8 Queens

- `boolean queens(int board[], int nextColumnToTry);`
  - Base Case: The problem has already been solved
  - When is the problem solved?
    - `nextColumnToTry ≥ board.length` – we've placed all of the pieces
    - Print the board

#### 15-12: Backtracking - 8 Queens

- `boolean queens(int board[], int nextColumnToTry);`
  - Recursive case:
    - For each of the `board.length` positions we could place the next queen, if the location is valid
      - Place the queen
      - Try to recursively solve the rest of the problem. If it succeeds, stop and return true
      - Otherwise, try next location
    - If no locations worked, return false

#### 15-13: Backtracking - 8 Queens

```
public static boolean queens(int board[], int nextColumn)
{
    if (nextColumn == board.length)
    {
        printBoard(board);
        return true;
    }
    for (int i = 0; i < board.length; i++)
    {
        board[nextColumn] = i;
        if (legal(board, nextColumn + 1))
            if (queens(board, nextColumn+1))
                return true;
    }
    return false;
}
```

#### 15-14: Backtracking - 8 Queens

```
public static boolean legal(int board[], int colsPlaced)
{
    boolean legal = true;
    for (int i=0; i < colsPlaced; i++) {
        for (int j=i+1; j<colsPlaced; j++) {
            if ((board[i] == board[j]) ||
                (Math.abs(i-j) == Math.abs(board[i]-board[j])))
            {
                legal = false;
            }
        }
    }
    return legal;
}
```

#### 15-15: Backtracking - Mazes

- Solving Mazes