16-0: **Recursion Review**

- Steps to solving a problem recusively
  - What is a small verson of the problem, that you can solve immediately?
  - How can you make the problem smaller?
  - Given a solution to the smaller problem, how can you solve the original problem?

16-1: **Recursion – NumDigits**

- Write a method that returns the number of base-10 digits in a number
  `int numDigits(int n)`
  - `numDigits(20201) == 5`
  - `numDigits(34) == 2`
  - `numDigits(3) == 1`
  - `numDigits(3050060) == 7`
- 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?

16-2: **Recursion – NumDigits**

```
int numDigits(int n)
{
   if (n < 10)
   {
      return 1;
   }
   else
   {
      return 1 + numDigits(n / 10);
   }
}
```

16-3: **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?

16-4: **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? 16-5: **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? 16-6: **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") ==> "nop ie"
```

16-7: **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));
  }
}
```

16-8: **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

16-9: **Backtracking - 8 Queens**

- 8 Queens on whiteboard

16-10: **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

16-11: **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 $0 \leq i <$`nextColumnToTry`

16-12: **Backtracking - 8 Queens**

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

16-13: **Backtracking - 8 Queens**

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

16-14: **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

16-15: **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;
}
```

16-16: **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;
}
```

16-17: **Recursion Problems**

- Write a recursive function that returns the sum of all of the base-10 digits in a number. So sumDigits(341) would return 8 (3 + 4 + 1 = 9), and sumDigits 23412 would return 12 (2 + 3 + 4 + 1 + 2 = 12)
- Write a recursive function that returns the smallest value in the first `size` elements of an array of integers
    - `int minimum(int A[], int size)`