Thinking Recursively
The 2013 Pulitzer Prizes

“Awarded to the Sun Sentinel, Fort Lauderdale, FL, for its well documented investigation of off-duty police officers who recklessly speed and endanger the lives of citizens, leading to disciplinary action and other steps to curtail a deadly hazard.”

http://www.pulitzer.org/citation/2013-Public-Service
Announcements

• YEAH hours (Your Early Assignment Help) hours for Assignment 2 this Thursday, April 18, in Gates B12 from 5:30PM – 6:30PM.

• Optional, but recommended!
Thinking Recursively
Recursive Problem-Solving

if (problem is sufficiently simple) {
    Directly solve the problem.
    Return the solution.
} else {
    Split the problem up into one or more smaller problems with the same structure as the original.
    Solve each of those smaller problems.
    Combine the results to get the overall solution.
    Return the overall solution.
}
```c
int digitalRoot(int value);
int sumOfDigits(int value);

int sumOfDigits(int value) {
    if (value == 0) {
        return 0;
    } else {
        return sumOfDigits(value / 10) + (value % 10);
    }
}

int digitalRoot(int value) {
    if (value < 10) {
        return value;
    } else {
        return digitalRoot(sumOfDigits(value));
    }
}
```
int maximumCoverageFor(Vector<int>& populations) {
    if (populations.size() == 0) {
        return 0;
    } else if (populations.size() == 1) {
        return populations[0];
    } else {
        Vector<int> allButFirst = tailOf(populations);
        Vector<int> allButFirstTwo = tailOf(allButFirst);

        return max(maximumCoverageFor(allButFirst),
                    populations[0] +
                    maximumCoverageFor(allButFirstTwo));
    }
}
What's Going On?

• Recursion solves a problem by continuously simplifying the problem until it becomes simple enough to be solved directly.

• The **recursive decomposition** makes the problem slightly simpler at each step.

• The **base case** is what ultimately makes the problem solvable – it guarantees that when the problem is sufficiently simple, we can just solve it directly.
The Towers of Hanoi Problem
Towers of Hanoi
Towers of Hanoi

Move this tower...
Towers of Hanoi

Move this tower...
...to this spindle.
Towers of Hanoi

Move this tower...

...to this spindle.
Towers of Hanoi
Towers of Hanoi
Towers of Hanoi
Towers of Hanoi
Towers of Hanoi
Towers of Hanoi
Towers of Hanoi
Towers of Hanoi
Towers of Hanoi
Towers of Hanoi
Towers of Hanoi
Towers of Hanoi
Towers of Hanoi
Towers of Hanoi
Towers of Hanoi
Towers of Hanoi
Towers of Hanoi
Towers of Hanoi
Towers of Hanoi
Towers of Hanoi
Towers of Hanoi
Towers of Hanoi
Towers of Hanoi
Towers of Hanoi
Towers of Hanoi
Towers of Hanoi
Towers of Hanoi
Towers of Hanoi
Towers of Hanoi
Towers of Hanoi
Towers of Hanoi
Towers of Hanoi
Towers of Hanoi
Towers of Hanoi
Towers of Hanoi
Towers of Hanoi
Solving the Towers of Hanoi
Solving the Towers of Hanoi

This disk...

...needs to get over here.

...needs to get over here.
Solving the Towers of Hanoi

This disk...

...needs to get over here.

...needs to get over here.
Solving the Towers of Hanoi

A

B

C

This disk...

...needs to get over here.

...needs to get over here.
Solving the Towers of Hanoi
Solving the Towers of Hanoi
Solving the Towers of Hanoi

A

B

C
Solving the Towers of Hanoi

This disk...

...needs to get over here.
Solving the Towers of Hanoi

This disk...

...needs to get over here.

This disk...

...needs to get over here.
Solving the Towers of Hanoi

A

B

C

This disk...

...needs to get over here.

...needs to get over here.
Solving the Towers of Hanoi

A

This disk...

B

...needs to get over here.

C

...needs to get over here.
Solving the Towers of Hanoi

A

B

C
Solving the Towers of Hanoi
Solving the Towers of Hanoi

This disk...

...needs to get over here.

This disk...

...needs to get over here.
Solving the Towers of Hanoi

A  

B  

C  

This disk...

...needs to get over here.
Solving the Towers of Hanoi

A

This disk...

B

...needs to get over here.

C

...needs to get over here.
Solving the Towers of Hanoi

This disk...

...needs to get over here.

...needs to get over here.
The Base Case

- We need to find a very simple case that we can solve directly in order for the recursion to work.
- If the tower has size one, we can just move that single disk from the source to the destination.
And now, the solution...
void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}
```c
int main() {
    moveTower(3, 'a', 'c', 'b');
}
```
```c
int main() {
    moveTower(3, 'a', 'c', 'b');
}
```
```c
int main() {
    moveTower(3, 'a', 'b', 'c');
}

void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}
```
```c
int main() {
    moveTower(3, 'a', 'b', 'c');
}

void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}
```
```c
int main() {
    moveTower(3, 'a', 'b', 'c');
}

void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}
```
void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}
```c
int main() {
    moveTower(3, 'a', 'b', 'c');
}

void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}
```
```c
void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}
```
int main() {
}

void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}

int main() {
    moveTower(3, 'a', 'b', 'c');
}
```c
int main() {
    moveTower(3, 'a', 'b', 'c');
}

void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}
```

The diagram illustrates the Tower of Hanoi problem. Initially, the disks are on pole A. The goal is to move all disks to pole B, following the rules:
- Only one disk can be moved at a time.
- A disk can only be moved if it is the top disk of a pole.
- A larger disk cannot be placed on top of a smaller disk.

In the diagram:
- The red disk is on pole A.
- The yellow disk is on pole B.
- The green disk is on pole C.

The current configuration shows that the disks need to be moved from pole A to pole B, with pole C as the temporary holding pole.
```c
int main() {
    moveTower(3, 'a', 'b', 'c');
}

void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}
```

Diagram:
- A: 3 disks: green, yellow, red
- B: Empty
- C: Empty

- Move the top disk from A to C
- Move the middle disk from A to B
- Move the top disk from C to B
- Move the middle disk from B to C
- Move the top disk from B to A
- Move the middle disk from C to A
- Move the top disk from B to C
```c
int main() {
    moveTower(3, 'a', 'b', 'c');
}

// Move tower function
void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}
```
void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}

int main() {
    moveTower(3, 'a', 'b', 'c');
}

int main() {
    moveTower(3, 'a', 'b', 'c');
}
int main() {
    moveTower(3, 'a', 'b', 'c');
}

void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}

void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}

void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}

void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}
```c
int main() {
    moveTower(3, 'a', 'b', 'c');
}

void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}
```

The diagram shows the movement of disks from one tower to another, with the following configuration:

- **n = 2**: Two disks are moved.
- **from = a**: Disk moves from tower 'a'.
- **to = b**: Disk moves to tower 'b'.
- **temp = c**: Tower 'c' is used as a temporary location.

The diagram illustrates the movement of disks in the Tower of Hanoi puzzle, where disks are moved from one tower to another, following specific rules.

- Disk arrangement: A (2 disks), B (1 disk), C (1 disk).
```c
void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}
```

The diagram shows a state of moving a tower of disks from A to B using C as the temporary peg. Initially, there are 2 disks on A. After the first move, disk 1 is moved to B, leaving disk 2 on A. Then, disk 2 is moved to C, moving disks 1 and 2 back to A. Finally, disk 1 is moved to B, completing the move.
```c
int main() {
}

void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}
```

The diagram shows the state of the towers after moving the disks. The disks on the leftmost tower are moved to the rightmost tower, with a temporary tower in between. The diagram illustrates the recursive nature of the moveTower function.
int main() {
    moveTower(3, 'a', 'b', 'c');
}

void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}

void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}

n 2 from a to b temp c

A   B   C
```c
int main() {
    moveTower(3, 'a', 'b', 'c');
}

void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}
```
```c
int main() {
    moveTower(3, 'a', 'b', 'c');
}

void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}
```
```c
int main() {
    moveTower(3, 'a', 'b', 'c');
}

void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}
```
```c
int main() {
    moveTower(3, 'a', 'b', 'c');
}

void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n – 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n – 1, temp, to, from);
    }
}
```

- **Disks**: 3
- **Source**: C
- **Destination**: B
- **Temp**: A

---

A | B | C
---|---|---
|   |   |   |
int main() {
    moveTower(3, 'a', 'b', 'c');
}

void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}

# Tower of Hanoi Diagram

**Initial State:**
- A: ![Green Disk](#)
- B: ![Red Disk](#) on top of ![Yellow Disk](#)
- C: Empty

**Move 1:**
- Move the single disk from A to B

**Final State:**
- A: Empty
- B: ![Yellow Disk](#) on top of ![Red Disk](#)
- C: Empty
```c
int main() {
    moveTower(3, 'a', 'b', 'c');
}

void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n – 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n – 1, temp, to, from);
    }
}
```
```c
int main() {
    moveTower(3, 'a', 'b', 'c');
}

void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}
```
```c
int main() {
    moveTower(3, 'a', 'b', 'c');
}

void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}
```

The function `moveTower` is used to move the tower of disks from one peg to another using a temporary peg. The diagram shows the state of the pegs after moving the disks.
int main() {  
void moveTower(int n, char from, char to, char temp) {  
  if (n == 1) {  
    moveSingleDisk(from, to);  
  } else {  
    moveTower(n - 1, from, temp, to);  
    moveSingleDisk(from, to);  
    moveTower(n - 1, temp, to, from);  
  }  
}  

n 3 from a to c temp b
#854 3

```c
int main() {
    void moveTower(int n, char from, char to, char temp) {
        if (n == 1) {
            moveSingleDisk(from, to);
        } else {
            moveTower(n - 1, from, temp, to);
            moveSingleDisk(from, to);
            moveTower(n - 1, temp, to, from);
        }
    }

    // Example call to moveTower
    moveTower(2, 'b', 'c', 'a');
}
```
```c
int main() {
    moveTower(3, 'a', 'b', 'c');
}

void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}
```
```c
int main() {
    moveTower(3, 'a', 'b', 'c');
}

void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}
```
```c
int main() {
    moveTower(3, 'a', 'b', 'c');
}

void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}
```

**Diagram:**

```
A  B  C
```

- **A**
  - Brown disk

- **B**
  - Yellow disk
  - Red disk

- **C**
  - Green disk

**Move:**

- **n = 2** from **B** to **C**
  - Move yellow disk from **B** to **C**
  - Move red disk from **B** to **C** using **A** as temp
  - Move yellow disk from **B** to **C**
  - Move red disk from **A** to **B** using **C** as temp
  - Move yellow disk from **C** to **B**
```
int main() {
    moveTower(3, 'a', 'b', 'c');
}

void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n – 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n – 1, temp, to, from);
    }
}
```c
void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}
```
```c
int main() {
    moveTower(3, 'a', 'b', 'c');
}

void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}
```
```c
int main() {
    moveTower(3, 'a', 'b', 'c');
}

void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}
```
```c
int main() {
    void moveTower(int n, char from, char to, char temp) {
        if (n == 1) {
            moveSingleDisk(from, to);
        } else {
            moveTower(n - 1, from, temp, to);
            moveSingleDisk(from, to);
            moveTower(n - 1, temp, to, from);
        }
    }
}
```

Here is an illustration of the tower of Hanoi problem with 3 disks:

- **A** (Red) initially contains all disks.
- **B** (Yellow) is empty initially.
- **C** (Green) is empty initially.

Up to 2 disks are moved at a time, with temporary storage on B. The goal is to move all disks from A to C while maintaining the order.

```
   2   from   b   to   c   temp   a
   A   B   C
```
```c
void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}
```
int main() {

void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}

n 2  from b  to c  temp a

A B C
```c
int main() {
    moveTower(3, 'a', 'b', 'c');
}

void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}

n  2  from  b  to  c  temp  a
```
```c
int main() {
    moveTower(3, 'a', 'b', 'c');
}

void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}
```

```
void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}
```

```
void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}
```
```c
int main() {
    moveTower(3, 'a', 'b', 'c');
}

void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}

void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}
```

Diagram:
- A: 1 disk from A to C
- B: 2 disks from A to B
- C: 3 disks from B to C

- A: 1 disk from A to B
- B: 2 disks from B to C
- C: 3 disks from A to C
```c
int main() {
    moveTower(3, 'a', 'b', 'c');
}

void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n – 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n – 1, temp, to, from);
    }
}

void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n – 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n – 1, temp, to, from);
    }
}
```
```c
int main() {
    moveTower(3, 'a', 'b', 'c');
}

void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}
```
```c
int main() {
    moveTower(3, 'a', 'b', 'c');
}

void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}

void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}
```

Task: Move 3 disks from A to C using B as a temporary location.
```c
int main() {
    moveTower(3, 'a', 'b', 'c');
}

void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n – 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n – 1, temp, to, from);
    }
}
```

The diagram illustrates the movement of disks between three pegs (A, B, and C). The process starts with three disks on peg A and involves transferring them one by one to peg B while keeping the rule that a larger disk cannot be placed on top of a smaller one. The disks are transferred between pegs in sequence as indicated by the movement from A to B, B to C, and back to A, with the final placement on peg B.
```c
int main() {
    moveTower(3, 'a', 'b', 'c');
}
```
Emergent Behavior

- Even though each function call does very little work, the overall behavior of the function is to solve the Towers of Hanoi.
- It's often tricky to think recursively because of this emergent behavior:
  - No one function call solves the entire problem.
  - Each function does only a small amount of work on its own and delegates the rest.
Writing Recursive Functions

• Although it is good to be able to trace through a set of recursive calls to understand how they work, you will need to build up an intuition for recursion to use it effectively.

• You will need to learn to trust that your recursive calls – which are to the function that you are currently writing! – will indeed work correctly.
  • Eric Roberts calls this the “Recursive Leap of Faith.”

• Everyone can learn to think recursively. If this seems confusing now, don't panic. You'll start picking this up as we continue forward.
void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}
void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}
void moveTower(int n, char from, char to, char temp) {
    if (n == 0) {
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}
void moveTower(int n, char from, char to, char temp) {
    if (n != 0) {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}
```c
void moveTower(int n, char from, char to, char temp) {
    if (n != 0) {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}
```
Picking a Base Case

• When choosing base cases, you should always try to pick the absolute smallest case possible.

• The simplest case is often so simple that it appears silly.
  • Solve Towers of Hanoi with no disks.
  • Add up no numbers.
  • List all strings of no characters.

• This is a skill you'll build up with practice.
Parking Randomly

Car has length one
Parking Randomly
Parking Randomly

• Given a curb of length five, how many cars, on average, can park on the curb?

• We can get an approximate value through random simulation:
  • Simulate random parking a large number of times.
  • Output the average number of cars that could park.

• **Question**: How do we simulate parking cars on the curb?
Parking Randomly
Parking Randomly
Parking Randomly

0  x  x + 1  5
Parking Randomly
Parking Randomly

Place cars randomly in these ranges!
Parking Randomly

```cpp
int parkRandomly(double low, double high) {
    if (high - low < 1.0) {
        return 0;
    } else {
        double x = randomReal(low, high - 1.0);
        return 1 + parkRandomly(low, x) +
                parkRandomly(x + 1.0, high);
    }
}
```
The Parking Ratio

• The average number of cars that can be parked in a range of width $w$ for sufficiently large $w$ is approximately $0.7475972 \cdot w$

• The constant $0.7475972...$ is called Rényi's Parking Constant.

• For more details, visit http://mathworld.wolfram.com/RenyisParkingConstants.html.
So What?

• The beauty of our algorithm is the following recursive insight:

  *Split an area into smaller, independent pieces and solve each piece separately.*

• Many problems can be solved this way.
Next Time

- **Graphical Recursion**
  - How do you draw a self-similar object?

- **Exhaustive Recursion**
  - How do you generate all objects of some type?
  - Algorithms for subsets, permutations, and combinations.