Thinking Recursively
An Interesting Read

“How to Train Your Robot”

http://drtechniko.wordpress.com/2012/04/09/how-to-train-your-robot/

Teaching kids to program by having them program their parents!
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.
}
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));
    }
}
An Interesting Listen

This American Life:
“Take The Money and Run For Office”

http://www.thisamericanlife.org/radio-archives/episode/461/take-the-money-and-run-for-office
## Federal Campaign Limits

<table>
<thead>
<tr>
<th></th>
<th>To each candidate or candidate committee per election</th>
<th>To national party committee per calendar year</th>
<th>To state, district &amp; local party committee per calendar year</th>
<th>To any other political committee per calendar year[1]</th>
<th>Special Limits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Individual may give</td>
<td>$2,500*</td>
<td>$30,800*</td>
<td>$10,000 (combined limit)</td>
<td>$5,000</td>
<td>$117,000* overall biennial limit:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• $46,200* to all candidates</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• $70,800* to all PACs and parties[2]</td>
</tr>
</tbody>
</table>

# Federal Campaign Limits

<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Federal Campaign Limits</strong></td>
</tr>
<tr>
<td><strong>To each candidate or candidate committee per election</strong></td>
</tr>
<tr>
<td>Individual may give</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
Raising Money

• According to several sources (Bloomberg, the Wall Street Journal, Politico, etc.), the 2008 Presidential Election cost about $1.5 billion.

• How can you raise that much money from private donors?
Raising Money

$22,500

$7,500

$7,500

$7,500

$2,500 $2,500 $2,500

$2,500 $2,500 $2,500

$2,500 $2,500 $2,500

$2,500 $2,500 $2,500
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.

}
if (\(n \text{ is at most } 2,500\)) {
    
    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.
}
Recursive Problem-Solving

if (n is at most $2,500) {
    Donate $n
}

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.
}
Recursive Problem-Solving

if (n is at most $2,500) { 
    Donate $n
} else {
    Find three other people.
    Tell them each to raise $n / 3.
}
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

A

B

C
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

A

B

C

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

A  B  C

[Diagram of the Towers of Hanoi]
Solving the Towers of Hanoi

A

B

C
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

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.
Solving the Towers of Hanoi
Solving the Towers of Hanoi
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

This disk...  ...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

This disk...

...needs to get over here.

...needs to get over here.
The Recursive Decomposition

- To get a tower of $N+1$ disks from spindle X to spindle Y, using Z as a temporary:
  - **Recursively** move the top $N$ disks from spindle X to spindle Z, using Y as a temporary.
  - Move the $N+1$st disk from X to Y.
  - **Recursively** move the $N$ disks from spindle Z to spindle Y, using X as a temporary.
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);
    }
}
```

```
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 illustrates the movement of disks from column A to column C using column B as a temporary holding area. Initially, there are four disks on column A, with the largest disk at the bottom and the smallest at the top. The disks are moved following the rules of the Tower of Hanoi puzzle, with each step involving moving the smallest disk not on the target column to the temporary column, then moving the largest disk to the target column, and finally moving the smallest disk back from the temporary column to the target column.
```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);
    }
}
```

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

Diagram:
- Tower A: 3 disks
- Tower B: 2 disks
- Tower C: 1 disk
- Disks from A to B
- Disks from A to C
- Disks from B to C
- Disks from A to B
- Disks from A to C
- Disks from B 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);
    }
}
```

The function `moveTower` implements the Tower of Hanoi problem. The diagram shows the initial state of the disks on pole A and the target pole 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);
    }
}

table
\begin{tabular}{|c|c|c|}
\hline
n & 2 & from a to b temp c \\
\hline
\end{tabular}

Diagram
```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
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 moveSingleDisk(char from, char to) {
    // Implement moveSingleDisk logic
}
```

Diagram:
- Tower A with 3 disks: Red, Green, Yellow
- Tower B and Tower C are empty.
- Move from A to C using B as temp.
```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');
}
```
```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() {
    moveTower(3, 'a', 'b', '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);
    }
}

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);
    }
}

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);
    }
}

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**: 2 disks (Red and Green)
- **B**: Yellow disk
- **C**: Red disk

The tower moves disks from **A** to **B** using **C** as a temporary stack.
```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);
    }
}
```
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);
    }
}
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);
    }
}

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 1 from c to b 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);
    }
}
```

Diagram:

- A: Green
- B: Yellow
- C: Brown

Move sequence:

- Move disk 2 from A to B
- Move disk 1 from C to A
- Move disk 2 from B to C
- Move disk 1 from A to C
- Move disk 0 from B to C
```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);
        }
    }

    n 3 from a to c temp 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);
    }
}
```

A diagram showing the movement of three disks between pegs A, B, and C, with pegs labeled A, B, and C at the bottom.
```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**: 3
- **from**: a
- **to**: c
- **temp**: 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);
    }
}
```

The diagram shows the movement of the disks from one tower to another. Initially, there are two disks on tower B, and the goal is to move them to tower C using tower A as an auxiliary tower. The moves are made according to the rules of the Tower of Hanoi puzzle, where disks are moved one at a time, and no disk can be placed on top of a smaller disk.
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

Diagram:
- Tower A: B
- Tower B: C
- Tower C: A

Steps:
1. Move 2 disks from B to C (moveTower(2, B, C, A))
2. Move 1 disk from A to B (moveSingleDisk(A, B))
3. Move 2 disks from C to B (moveTower(2, C, A, B))
4. Move 1 disk from A to C (moveSingleDisk(A, C))
5. Move 2 disks from B to A (moveTower(2, B, C, A))
6. Move 1 disk from C to B (moveSingleDisk(C, B))
7. Move 2 disks from A to C (moveTower(2, A, B, C))
8. Move 1 disk from B to A (moveSingleDisk(B, A))
9. Move 2 disks from C to A (moveTower(2, C, B, A))
10. Move 1 disk from A to C (moveSingleDisk(A, C))

The disks are now in the correct order on Tower 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);
        }
    }
    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);
    }
}
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 1 from b to a temp 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);
    }
}
```

- **Diagram:**
  - **A:** Brown
  - **B:** Yellow and Red
  - **C:** Green

- **Move:**
  - **n:** 1
  - **From:** B
  - **To:** A
  - **Temp:** 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);
    }
}
```

The function `moveTower` recursively moves disks from one peg to another according to the rules of the Towers of Hanoi puzzle.
```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 tower at different stages:

- **Initial state**: All disks are on tower A.
- **First move**: Move the top disk from tower A to tower C, placing it on tower B.
- **Second move**: Move the second disk from tower A to tower B, placing it on tower C.
- **Final move**: Move the third disk from tower A to tower C, placing it on tower B.

After these moves, all disks are correctly placed on tower C, with tower B left empty and tower A initially having all disks.
```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);
    }
}
```

```
A
B
C

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);
    }
}
```

The diagram shows the state of the towers before and after the move of two disks:

- **Before:**
  - Tower A: Red disk
  - Tower B: Empty
  - Tower C: Empty

- **After:**
  - Tower A: Empty
  - Tower B: Red disk
  - Tower C: Red and Yellow disks
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);
    }
}

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);
    }
}
```

Diagram:
- **A**: Base
- **B**: Base
- **C**: Base
- **A** (Red):
  - **B** (Green):
    - **C** (Yellow):

The diagram shows the starting positions of the towers before the moveTower function is called with the parameters `n = 3`, `from = 'a'`, `to = 'c'`, and `temp = 'b'`. The towers start at the bottom and are moved one disk at a time, with the disk being moved from the start tower (A) to the destination tower (C) using a temporary tower (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
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');
}

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 b  temp c
```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);
    }
}
Parking Randomly

Car has length one
Parking Randomly

0
5
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

\[ 0 \leq x \leq x + 1 \leq 5 \]
Parking Randomly

Place cars randomly in these ranges!
```java
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, 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 $0.7475972 \ w$.
- The constant $0.7475972\ldots$ 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 the range into smaller, independent pieces and solve each piece separately.**

• Many problems can be solved this way.