Recursion

Section 9.6

Objectives

You will be able to

- write functions that call themselves in order to solve problems by dividing them into smaller but similar subproblems.

Recursive Function

A function that calls itself

- Why?
  - Many algorithms can be expressed more clearly and concisely in recursive form.
- "Divide and Conquer"
  - Solve a smaller problem.
  - Use result to solve original problem.

The Run-Time Stack

- Hidden data structure in every C program.
  - Maintained automatically.
  - Not directly accessible to the program.

- Each call to a function allocates some space on the run-time stack.
  - Called a *stack frame*
  - Function’s parameters are allocated here.
  - Function’s local variables are allocated here,
    - Except variables declared as static.
  - Return address is stored here.

The Run-Time Stack

- Each successive function call allocates a new stack frame.
  - Caller’s local variables are still on the stack in a lower stack frame.
  - Not visible to the newly called function.
  - Will still be there when the called function returns.

- This makes it possible for a function to call itself.

Recursive Calls

- If a function calls itself, there is a separate stack frame corresponding to each invocation.
  - arguments
  - local variables
  - return address

- A function can call itself multiple times without overwriting its local variables.
  - But not an unlimited number of times.
  - Eventually will run out of stack space!

A Very Simple Example

```
void countdown(int n)
{
    printf ("n = %d\n", n);
    n--;
    if (n >= 0)
    {
        countdown(n);        Call self
    }
}

int main ( )
{
    countdown(5);
    return 0;
}
```

Compile and Run

```
turnerr@login1:~/test
[turnerr@login1 test]$
[turnerr@login1 test]$ gcc -Wall countdown.c
[turnerr@login1 test]$ ./a.out
n = 5
n = 4
n = 3
n = 2
n = 1
n = 0
[turnerr@login1 test]$
[turnerr@login1 test]$
```

A Very Minor Change

```
void countdown(int n)
{
    //printf ("n = %d\n", n);
    n--;
    if (n >= 0)
    {
        countdown(n);
    }
    printf ("n = %d\n", n);
}

int main ( )
{
    countdown(5);
    return 0;
}
```

Now what is the output?

Compile and Run

```
turnerr@login1:~/test
[turnerr@login1 test]$
[turnerr@login1 test]$
[turnerr@login1 test]$ gcc -Wall countdown.c
[turnerr@login1 test]$ ./a.out
n = -1
n = 0
n = 1
n = 2
n = 3
n = 4
[turnerr@login1 test]$
```

Why did this happen?

Program countdown2.c

```
void countdown(int n)
{
    //printf ("n = %d\n", n);
    n--;
    if (n >= 0)
    {
        countdown(n);
    }
    printf ("n = %d\n", n);
}

int main ( )
{
    countdown(5);
    return 0;
}
```

Last invocation prints first

When it returns, next to last invocation prints.

...

First invocation prints last

Recursion can be tricky!

Inappropriate Use of Recursion

- A classic (and bad!) example of recursion is the factorial function.

Factorial Program

```
#include <stdio.h>
#include <assert.h>

int factorial(int n)
{
    assert (n >= 0);
    if (n <= 1)
    {
        return 1;
    }

    return n*factorial(n-1);
}
```

Factorial Program

```
int main()
{
    int n = -1;
    printf ("This program uses a recursive function\n");
    printf ("to compute N factorial\n\n");
    printf ("Enter N as a nonnegative integer: ");
    while (n < 0)
    {
        scanf("%d", &n);
        if (n < 0)
        {
             printf ("Invalid entry.  Please try again\n");
             while (getchar() != 'n');  // Clear keyboard input buffer
        }
    }

    printf ("%d factorial is %d\n", n, factorial(n));
    return 0;
}
```

Factorial Program in Action

```
turnerr@login1:~/test
[turnerr@login1 test]$
[turnerr@login1 test]$ gcc -Wall factorial.c
[turnerr@login1 test]$ ./a.out
This program uses a recursive function
to compute N factorial

Enter N as a nonnegative integer: 5
5 factorial is 120
[turnerr@login1 test]$
[turnerr@login1 test]$ ./a.out
This program uses a recursive function
to compute N factorial

Enter N as a nonnegative integer: 10
10 factorial is 3628800
[turnerr@login1 test]$
[turnerr@login1 test]$ ./a.out
This program uses a recursive function
to compute N factorial

Enter N as a nonnegative integer: 15
15 factorial is 2004310016
[turnerr@login1 test]$
[turnerr@login1 test]$ ./a.out
This program uses a recursive function
to compute N factorial

Enter N as a nonnegative integer: 20
20 factorial is -2102132736
[turnerr@login1 test]$
[turnerr@login1 test]$
```

Inappropriate Use of Recursion

- Why is this an inappropriate use of recursion?

  - The problem is not inherently recursive.
  - Can be solved just as easily with a loop.

- Prefer loops over recursion when it is feasible to do so.
  - Avoid the performance cost and conceptual subtleties of recursion.

An Inherently Recursive Calculation

- The Fibonacci series is defined as 1, 1, 2, 3, 5, ... where after the first two members, each member is the sum of the two preceding members.

Fibonacci main()

```
int main()
{
    int num, fibo;
    printf ("This program computes Fibonacci numbers\n\n");
    while (1)
    {
        printf ("Enter a (fairly small) integer: ");
        if (scanf("%d", &num) == 1)
        {
            fibo = Fibonacci(num);
            printf ("\nFibonacci(%d) = %d\n\n",  num, fibo);
        }
        else
        {
            printf ("Input error\n");
            printf ("Please try again\n\n");
            while (getchar() != '\n');  // Clear input buffer
        }
    }
}
```

Recursive Implementation

```
int Fibonacci (int n)
{
    if (n <= 2)
    {
        return 1;
    }

    return Fibonacci(n-1) + Fibonacci(n-2);
}
```

Try it!

Fibonacci Program in Action

```
turnerr@login1:~/test
[turnerr@login1 test]$
[turnerr@login1 test]$
[turnerr@login1 test]$ gcc -Wall fibo.c
[turnerr@login1 test]$ ./a.out
This program computes Fibonacci numbers

Enter a (fairly small) integer: 10

Fibonacci(10) = 55

Enter a (fairly small) integer: 20

Fibonacci(20) = 6765

Enter a (fairly small) integer: 30

Fibonacci(30) = 832040

Enter a (fairly small) integer: 40

Fibonacci(40) = 102334155

Enter a (fairly small) integer: 50

[turnerr@login1 test]$
```

Stopped with Ctrl-c

Measure Run Time

```
#include <time.h>
...
int main()
{
    int num, fibo;
    time_t start_time, end_time;
    printf ("This program computes Fibonacci numbers\n\n");

    while (1)
    {
        printf ("Enter a (fairly small) integer: ");
        if (scanf("%d", &num) == 1)
        {
            time(&start_time);
            fibo = Fibonacci(num);
            time(&end_time);
            printf ("\nFibonacci(%d) = %d\n",  num, fibo);
            printf ("%5.1f seconds\n\n", difftime(end_time, start_time));
        }
    }
```

Measure Run Time

turnerr@login1:~/test

```
[turnerr@login1 test]$
[turnerr@login1 test]$ gcc -Wall fibo.c
[turnerr@login1 test]$ ./a.out
This program computes Fibonacci numbers

Enter a (fairly small) integer: 30

Fibonacci(30) = 832040
   0.0 seconds

Enter a (fairly small) integer: 35

Fibonacci(35) = 9227465
   0.0 seconds

Enter a (fairly small) integer: 40

Fibonacci(40) = 102334155
   1.0 seconds

Enter a (fairly small) integer: 45

Fibonacci(45) = 1134903170
  10.0 seconds

Enter a (fairly small) integer:
[turnerr@login1 test]$
```

Ctrl-c

Loop Implementation

```
int Fibonacci (int n)
{
    int pred1 = 1;
    int pred2 = 1;
    int i, next;

    if (n <= 2)
    {
        return 1;
    }

    for (i = 3; i <= n; i++)
    {
        next = pred1 + pred2;    /* next is Fibonacci(i) */
        pred2 = pred1;
        pred1 = next;
    }
    return next;
}
```

Compare Implementations

- How does the run time for the loop implementation compare to that for the recursive implementation?

End of Example

Properly Formed Recursive Processes

- Every recursive process consists of two parts:

  1 One or more smallest base cases that are processed without recursion;

  2 A general method that reduces an arbitrary non-base case to one or more smaller cases

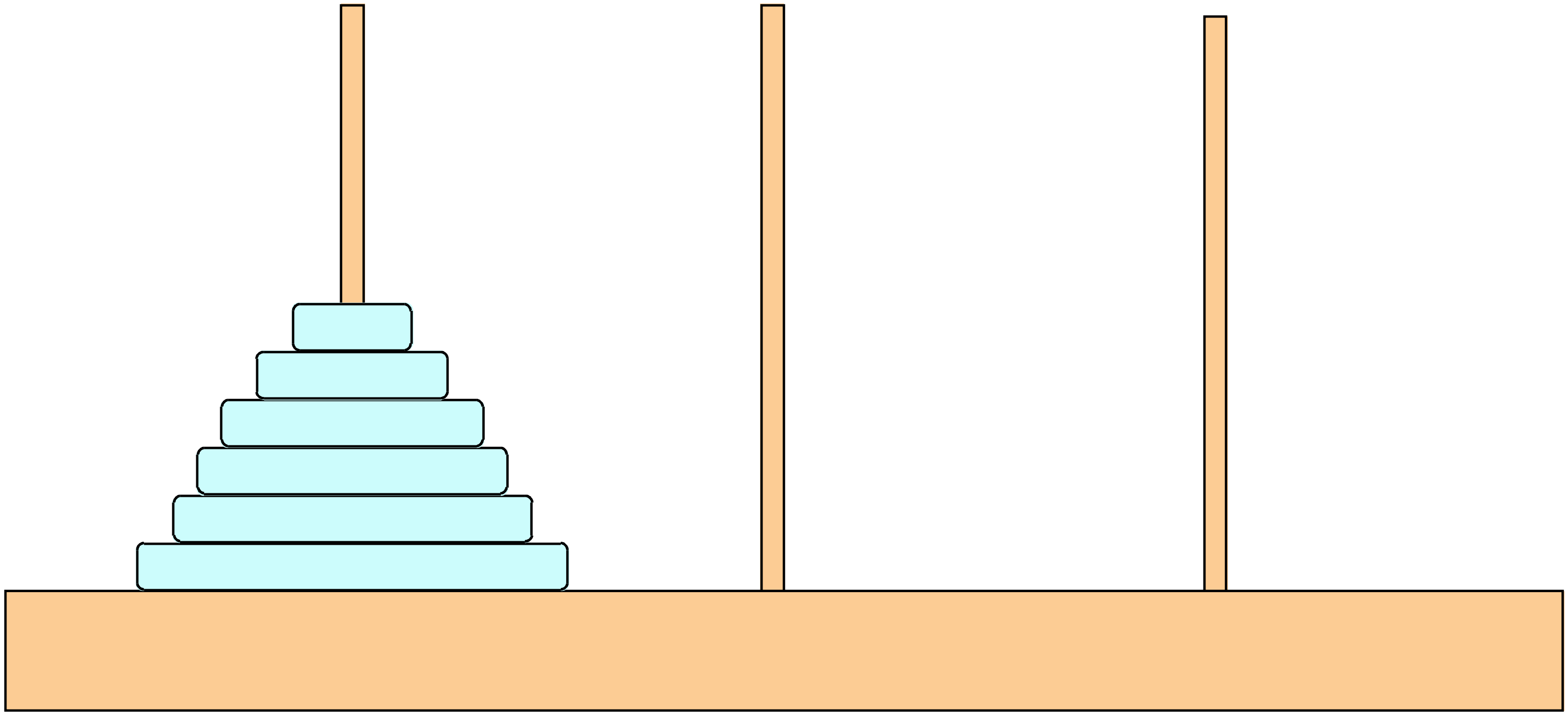
- The general method must be such that repeated application of the method will eventually reach base cases

Towers of Hanoi

- Brass platform with 3 diamond needles.
- On the first needle, 64 golden disks each slightly smaller than the one underneath it.

- **Task:**
  move all 64 disks to third needle, by moving one disk at a time from one needle to another so that a larger disk is never placed on top of a smaller one.

- **Story:**
  at the creation of the world, Buddhist priests were told to accomplish the task and that when it was done, the world would end.



Solution to the Towers of Hanoi problem

- **Restate problem:**

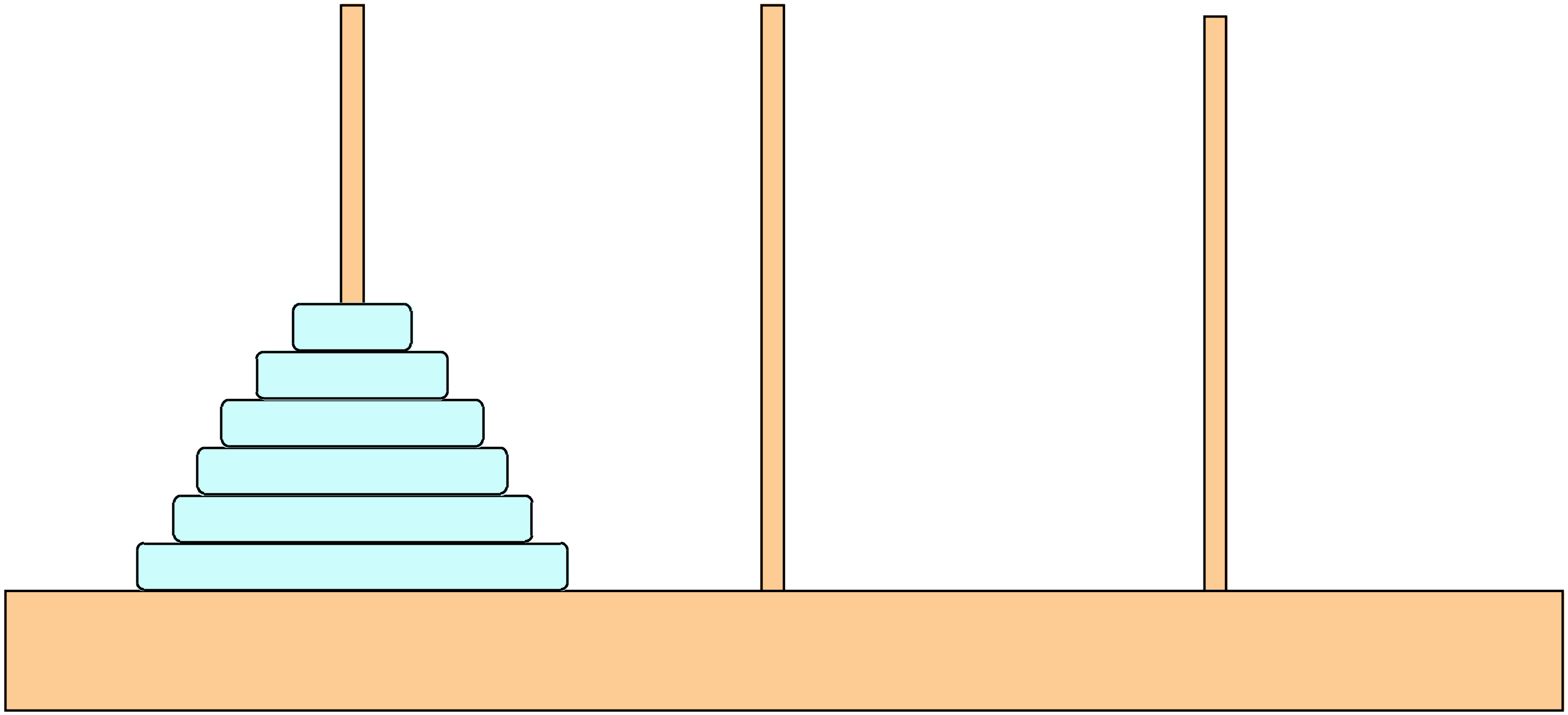
  move n disks from needle i to needle j using needle k as the auxiliary needle, where {i,j,k} = {1,2,3}

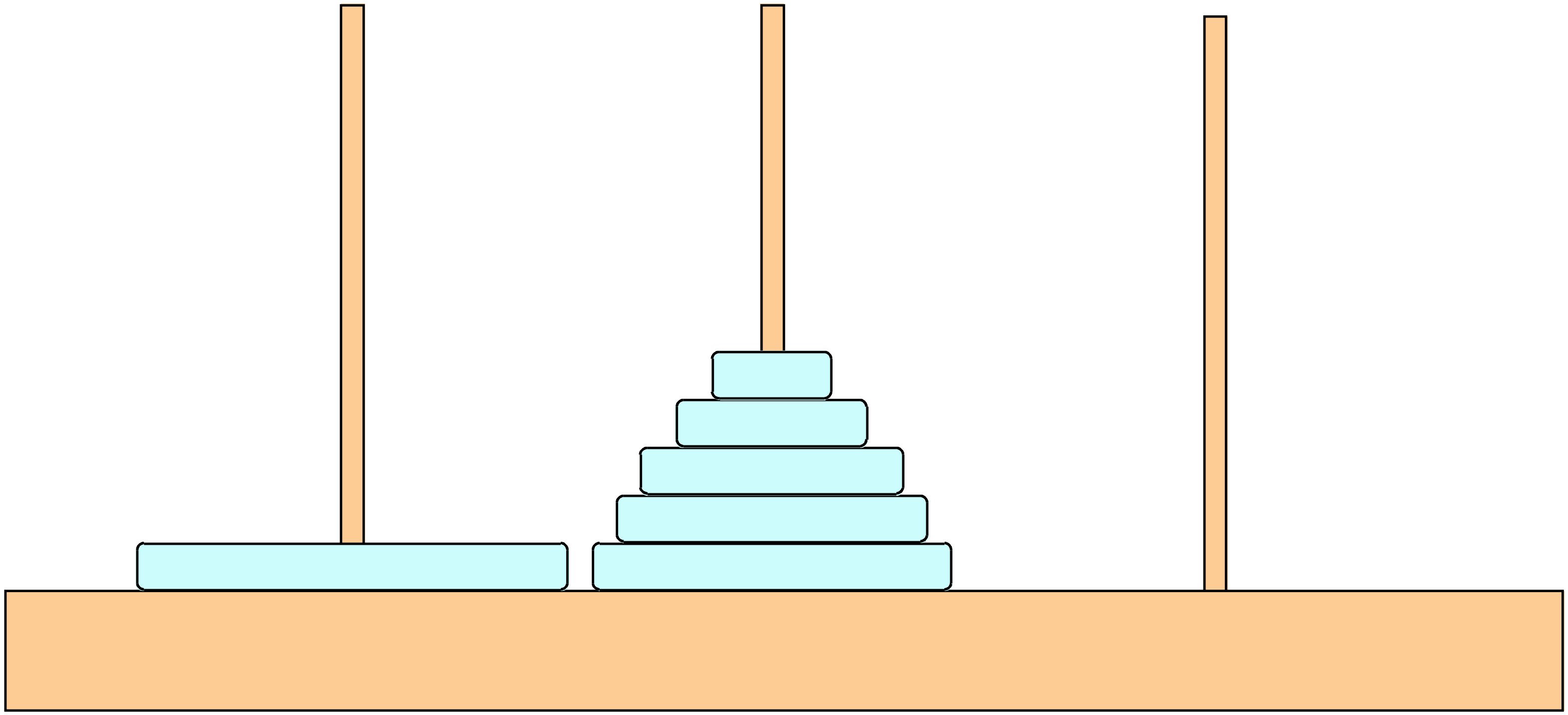
- **Recursive solution**

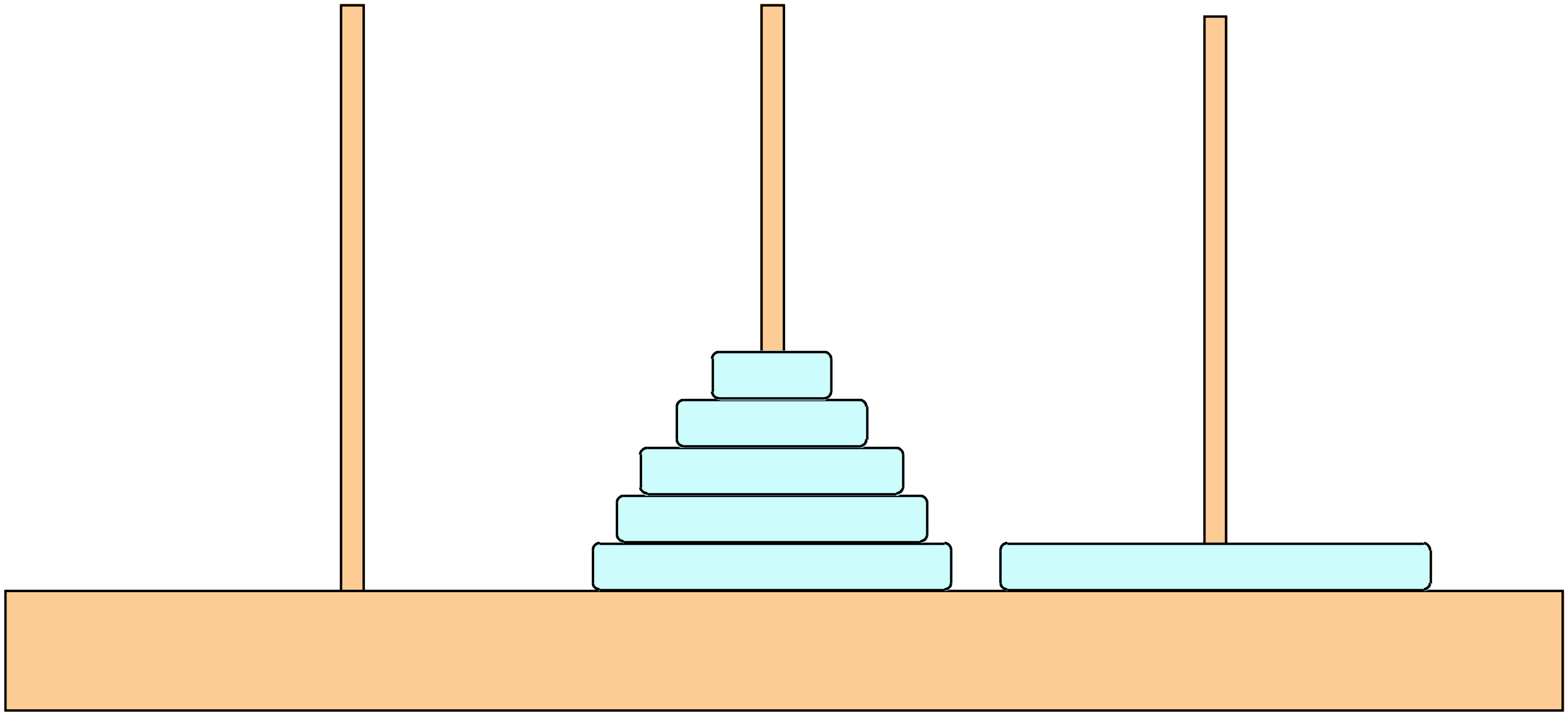
  1 Move the top n-1 disks from i to k using j as auxiliary

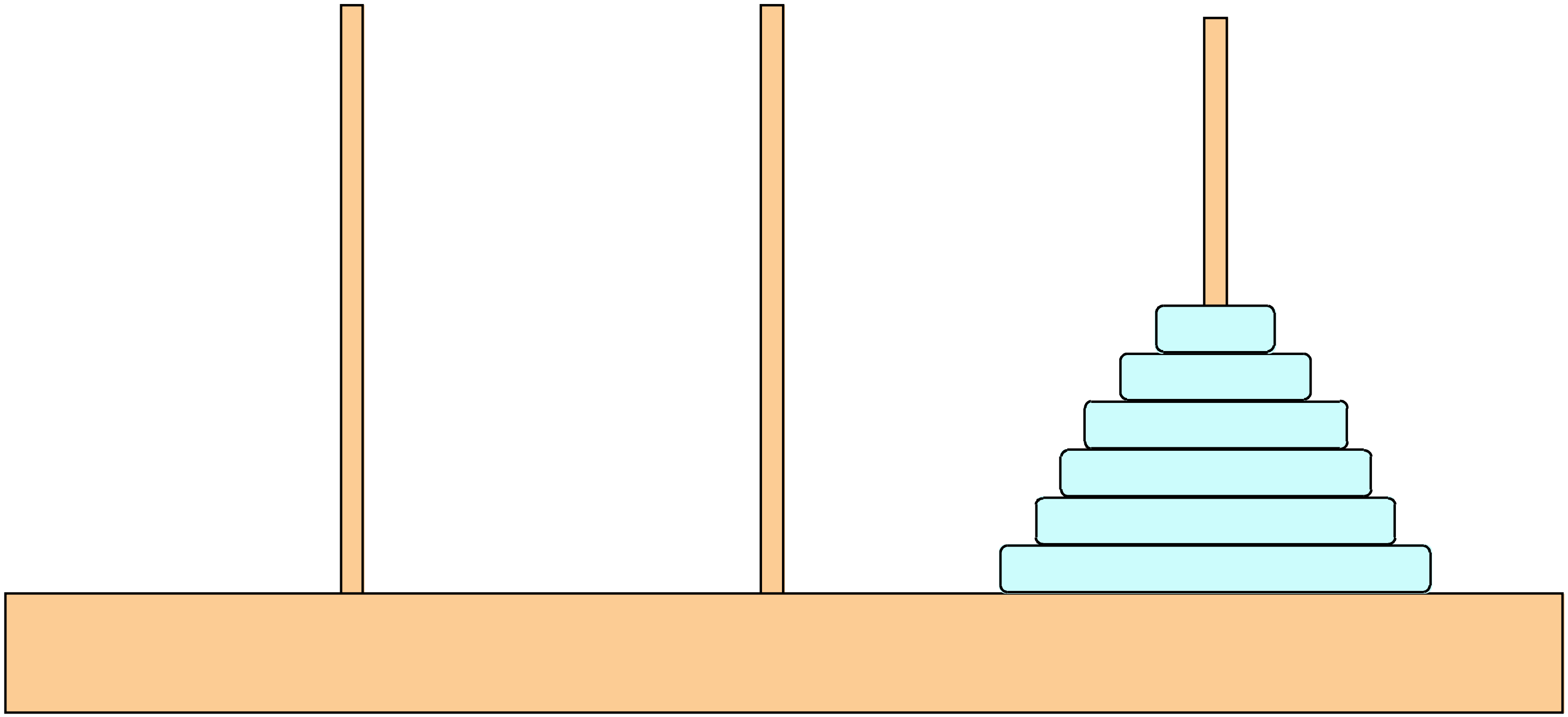
  2 Move single disk from i to j

  3 Move n-1 disks from k to j using i as auxiliary





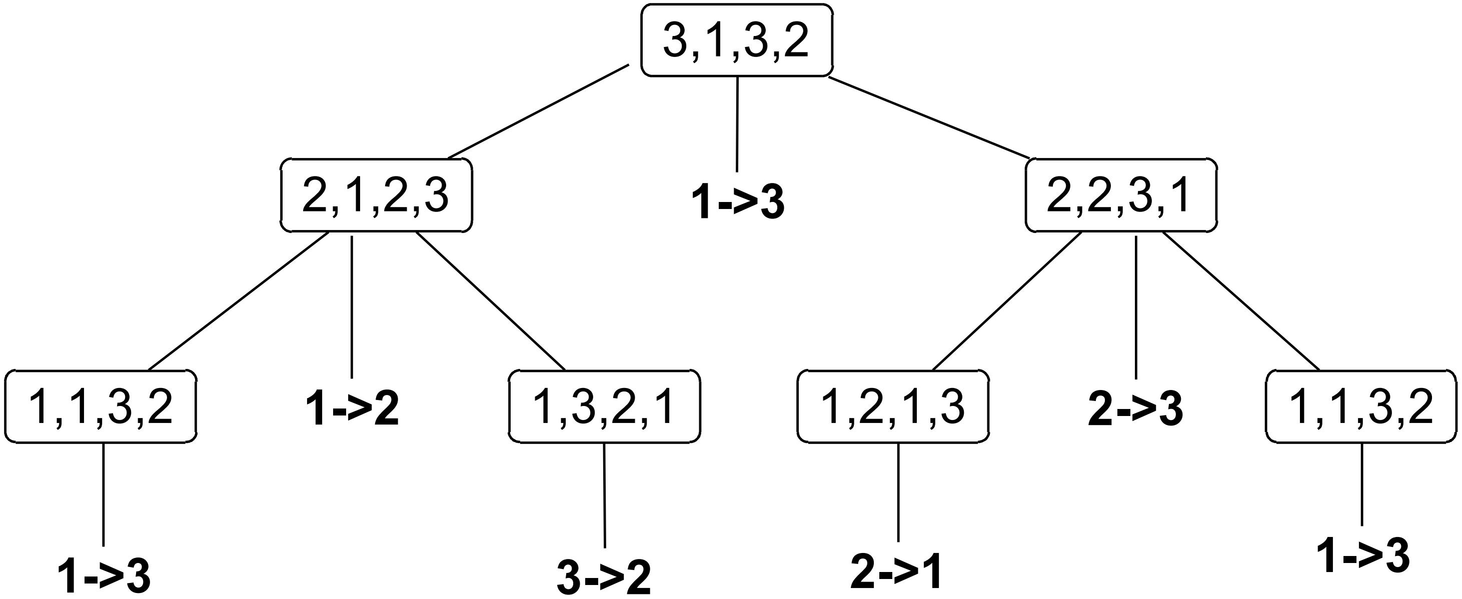




Code for recursive Towers of Hanoi

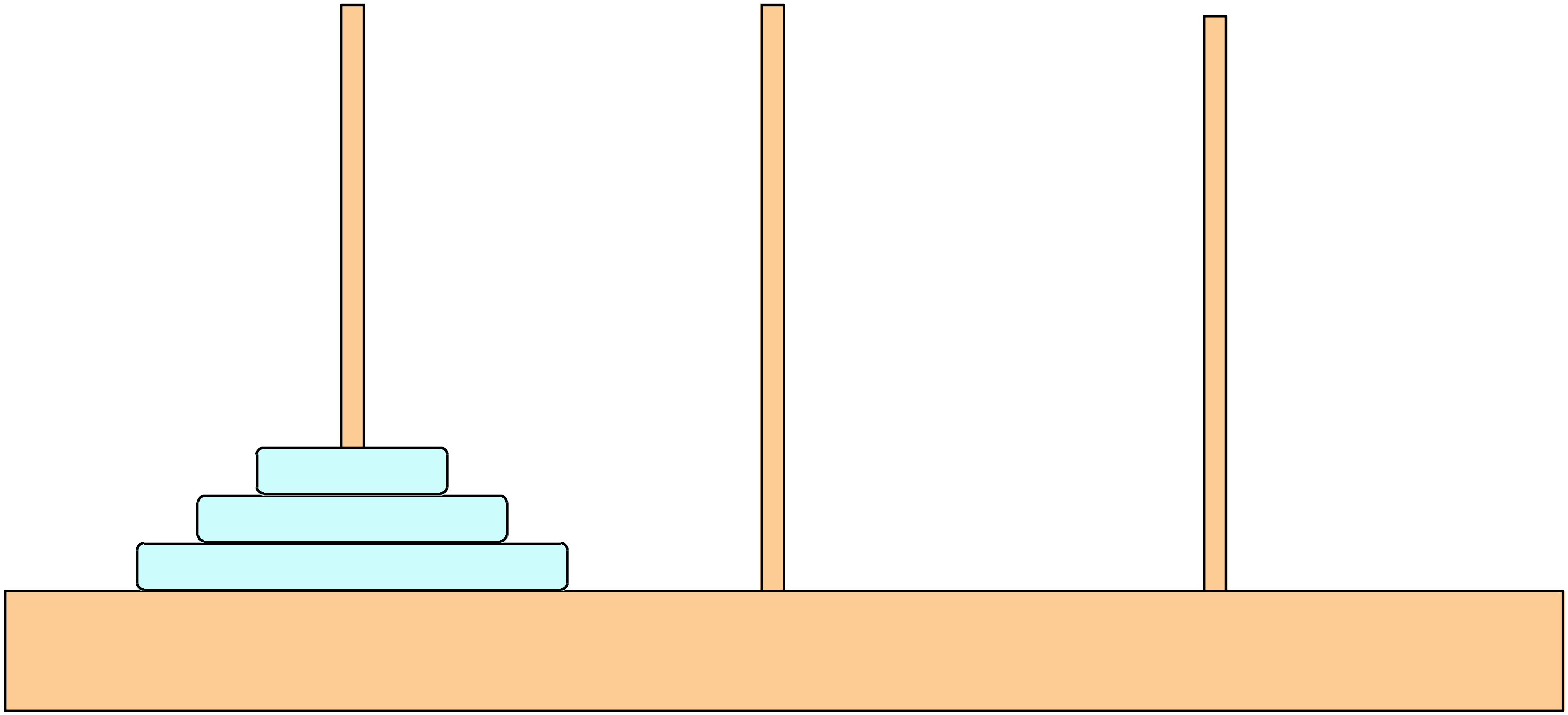
```
void Move(int count, int start, int finish, int temp)
{
    if (count == 0)
        return;
    if (count == 1)
        printf("%d -> %d\n", start, finish);
    else {
        Move(count-1, start,  temp, finish);
        printf("%d -> %d\n", start, finish);
        Move(count-1, temp, finish, start);
    }
}
```

Recursion Tree for Move(3,1,3,2)

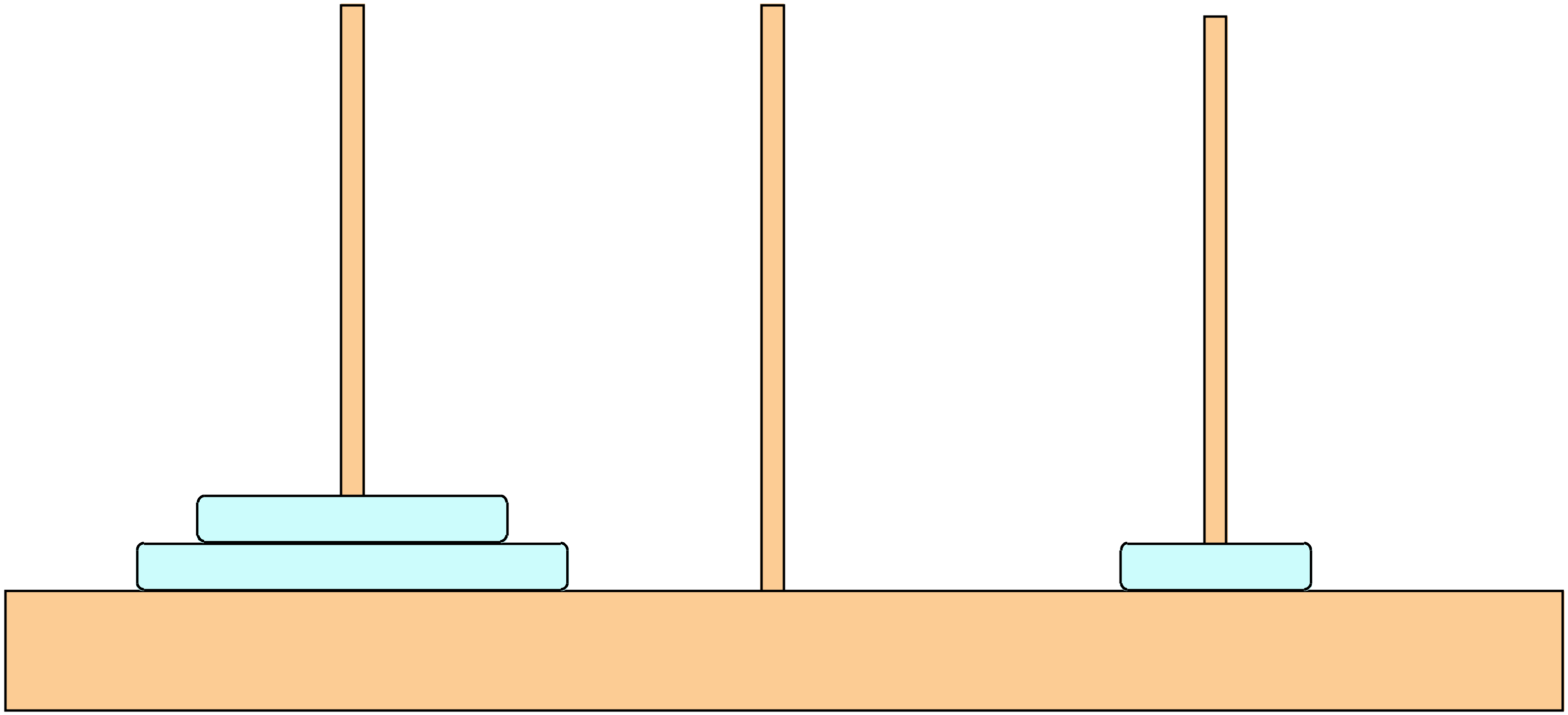


Instructions:

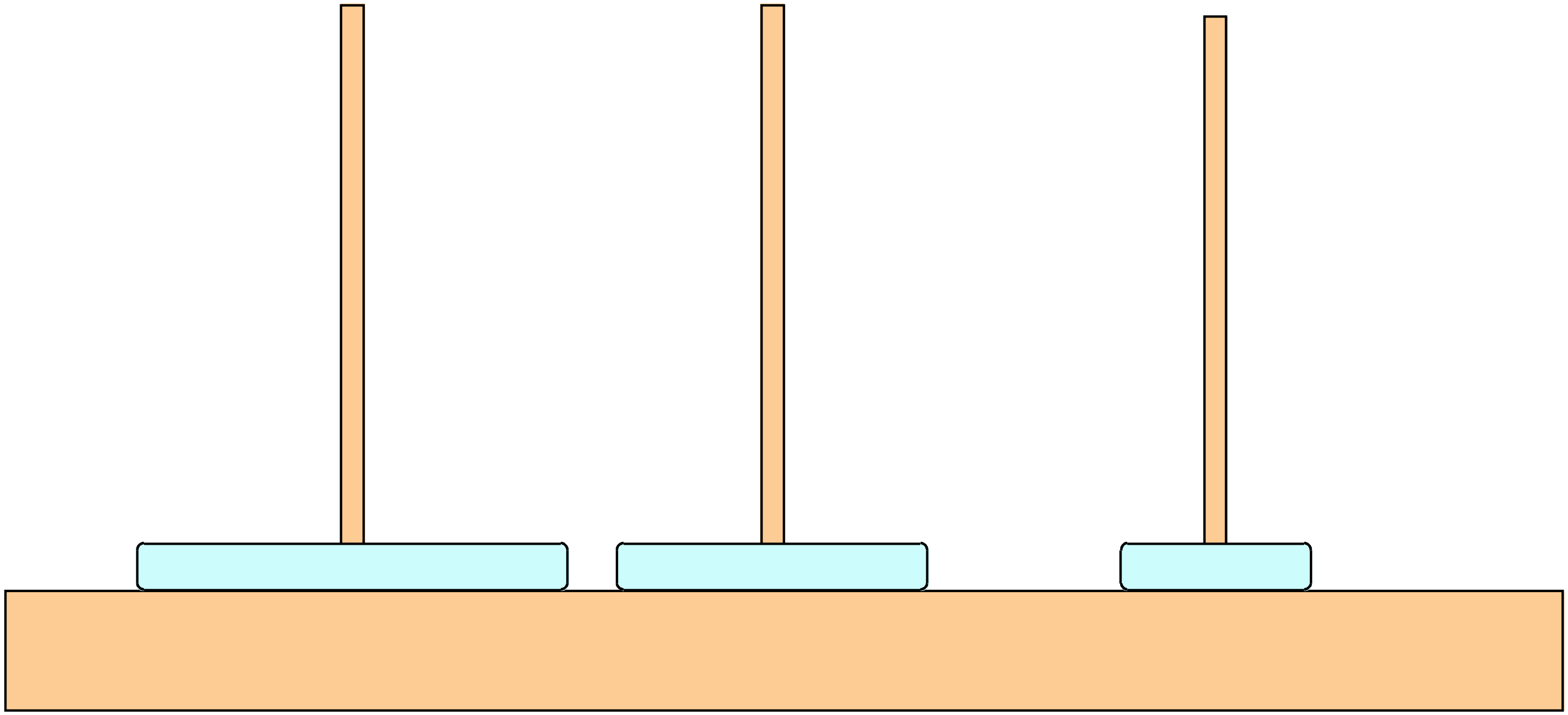
1->3 1->2 3->2 1->3 2->1 2->3 1->3



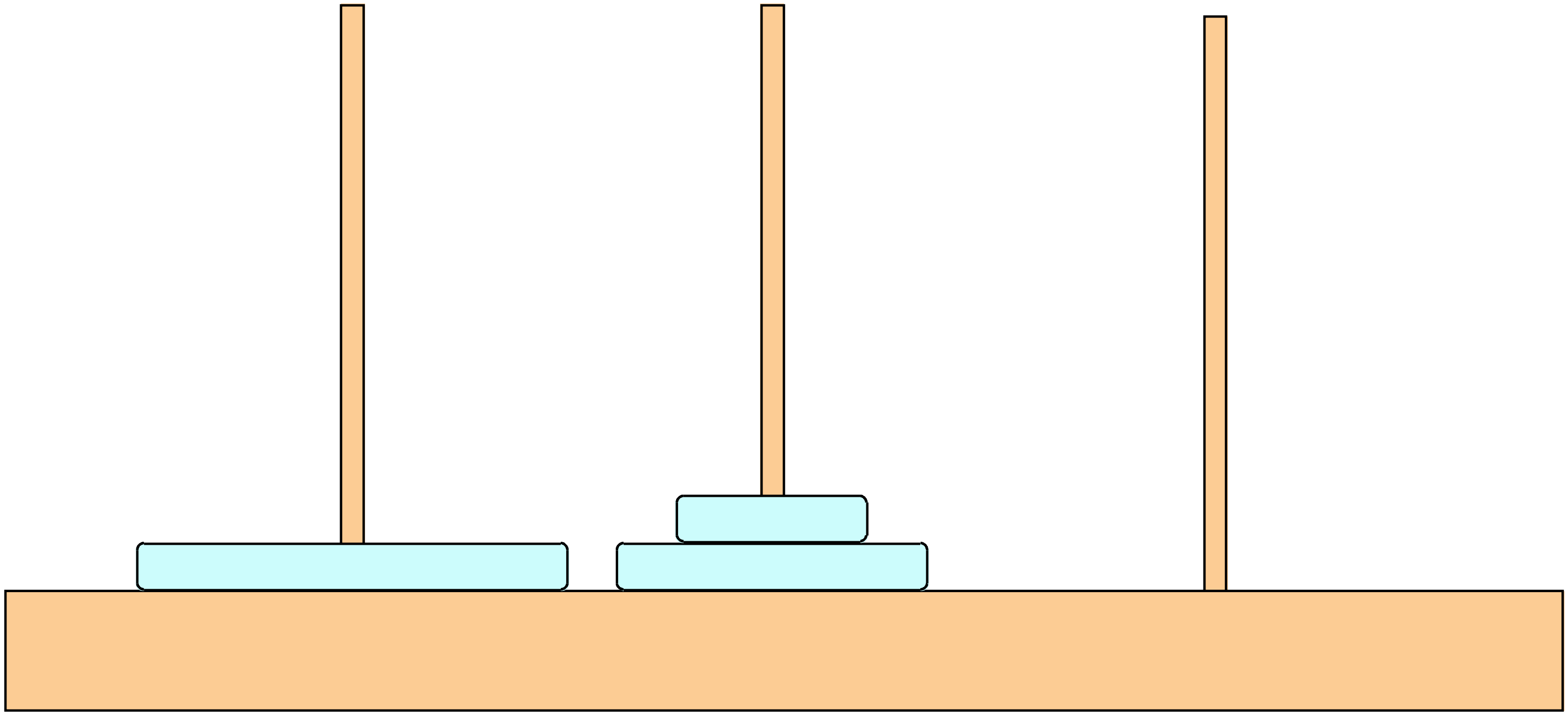
1->3



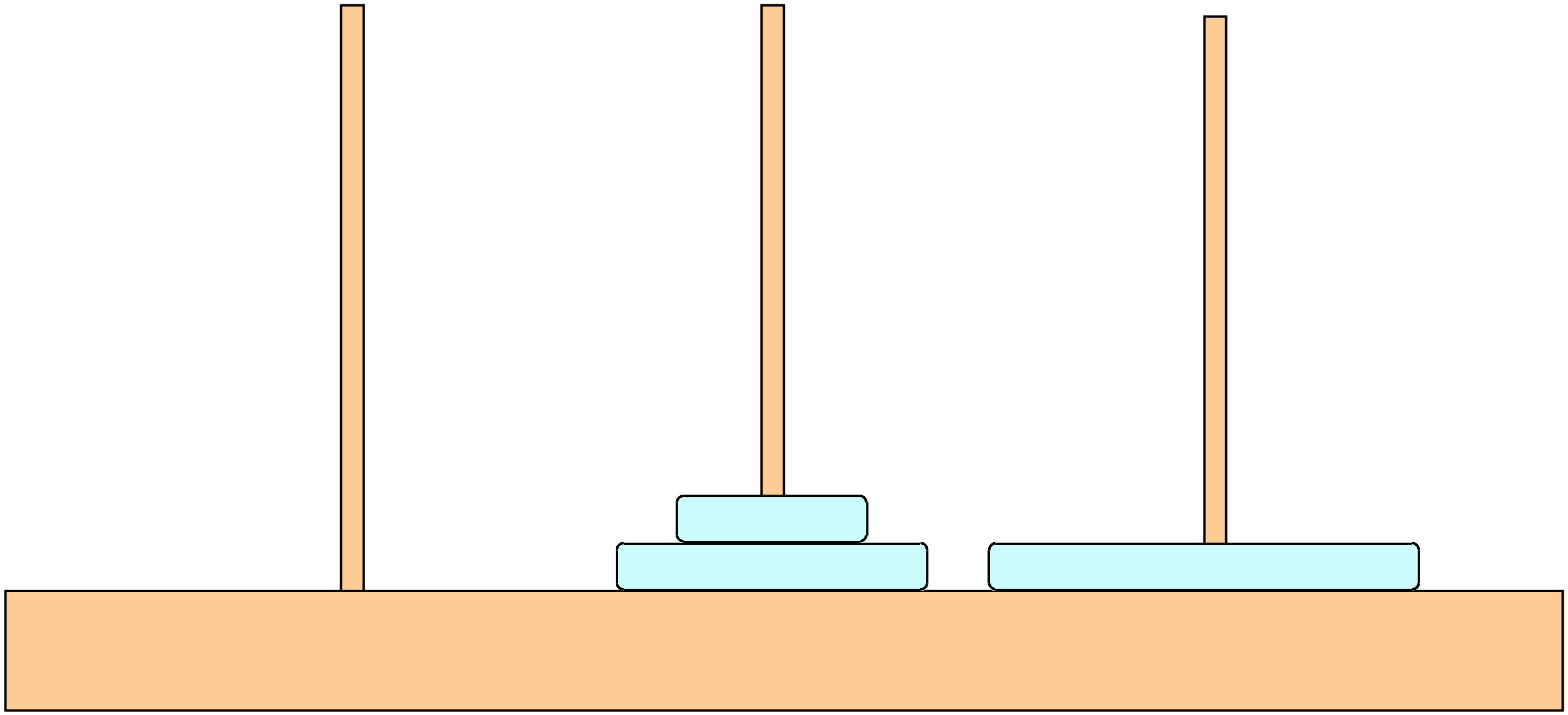
1->2



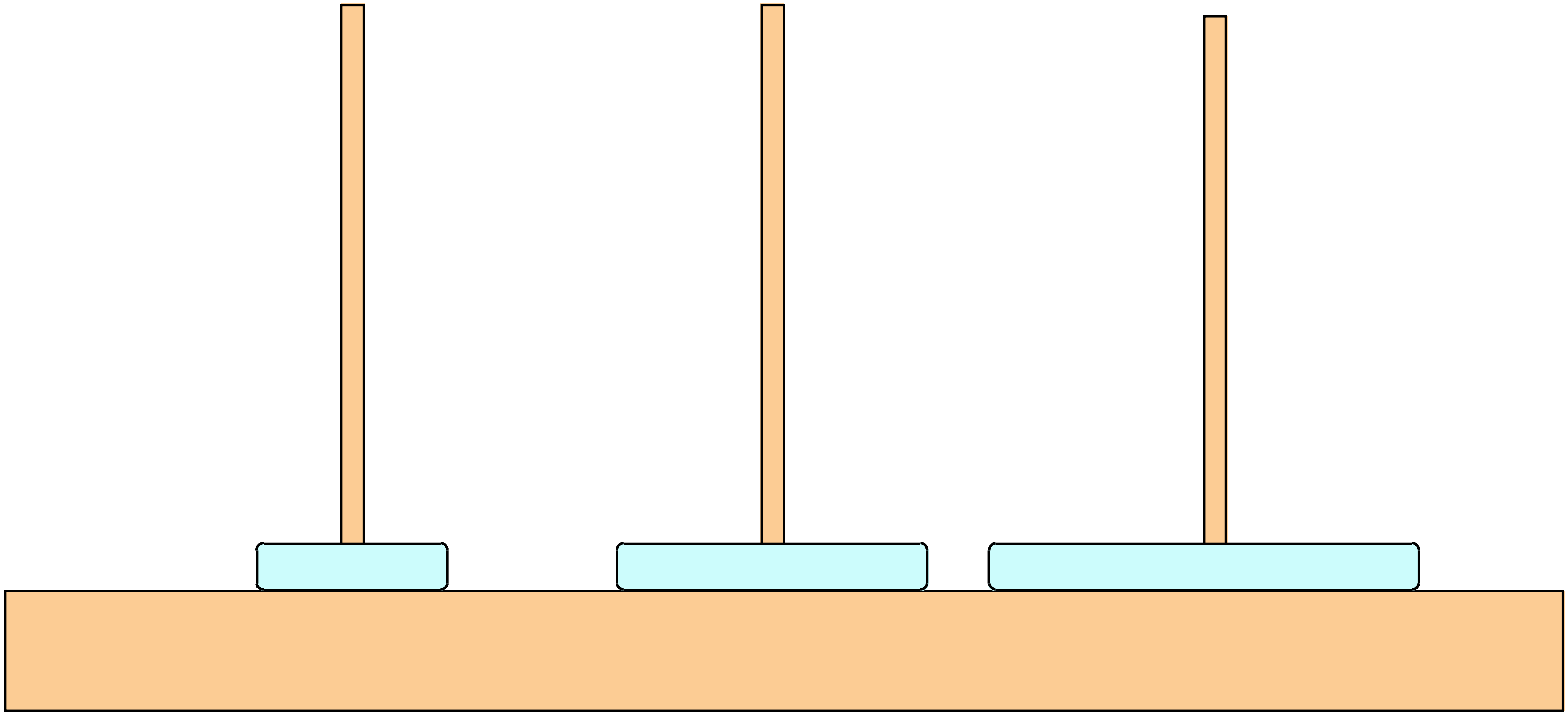
3->2



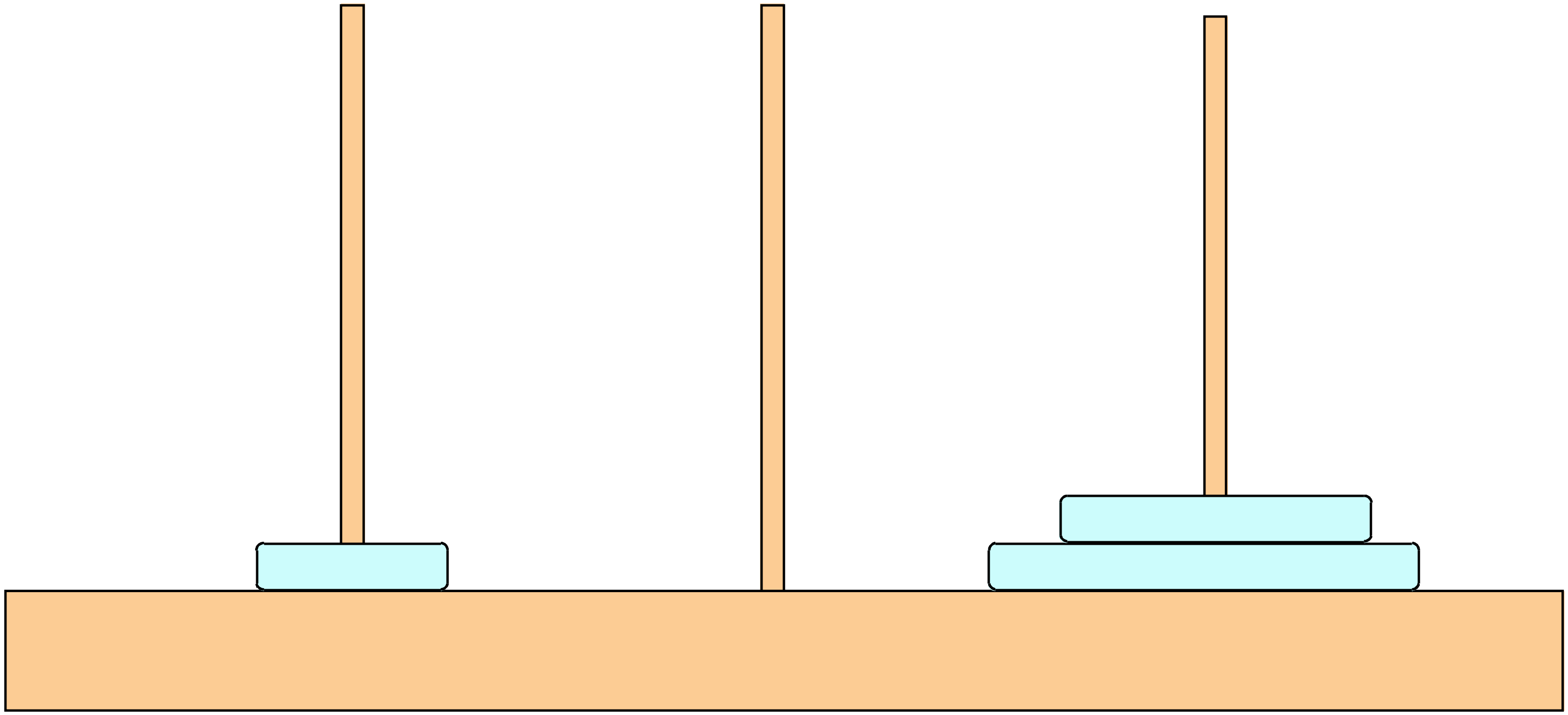
1->3



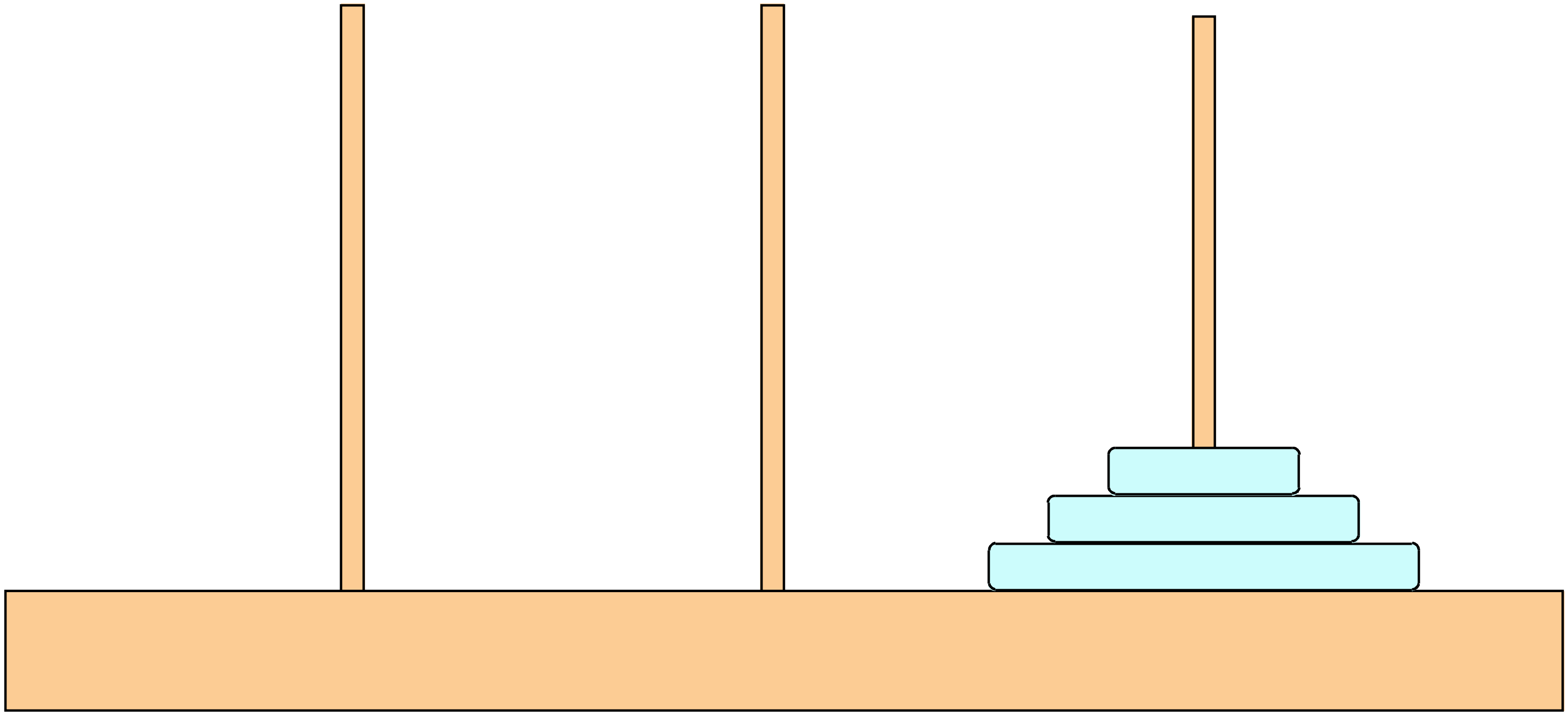
2->1



2->3



1->3



Analysis

- By analyzing the recursion tree for Move you will find that the number of moves is

$$1 + 2 + \ldots + 2^{n-1} = 2^n - 1$$

- For 64 disks, if the monks can move one disk each second, it will take approximately $5*10^{11}$ years
- This is 25 times the current estimated age of the universe!

Summary

- Recursion is one of the key tricks of the programming trade.
- "Divide and Conquer"
- Many algorithms can be expressed more clearly and concisely using recursion.
  - Typically at significant cost in terms of execution time and memory usage.