

# Design and Analysis of Algorithms

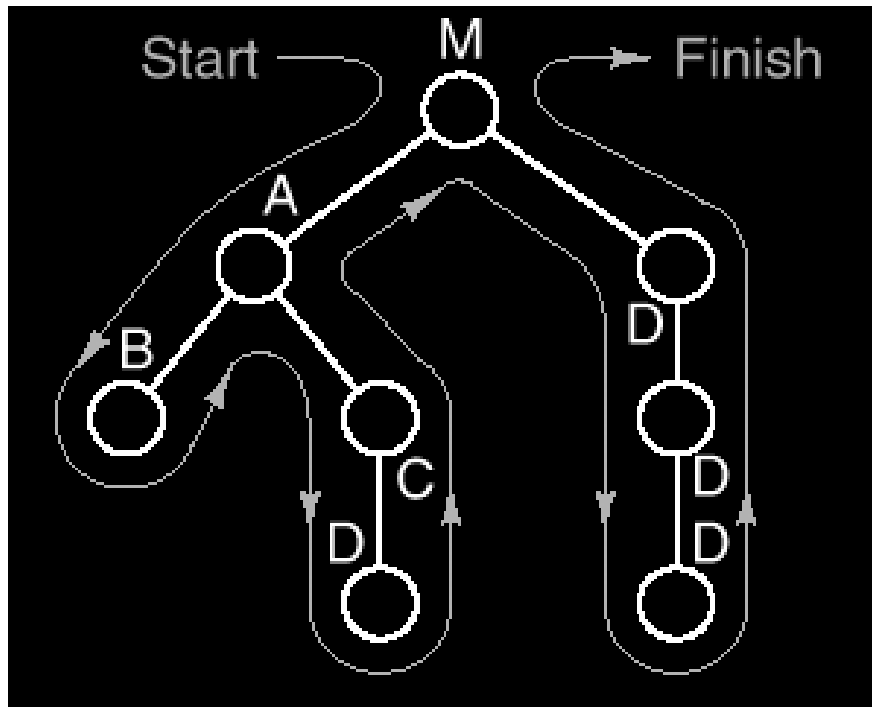
## *Lecture 4: Recursion*

# Lesson from Last Lecture

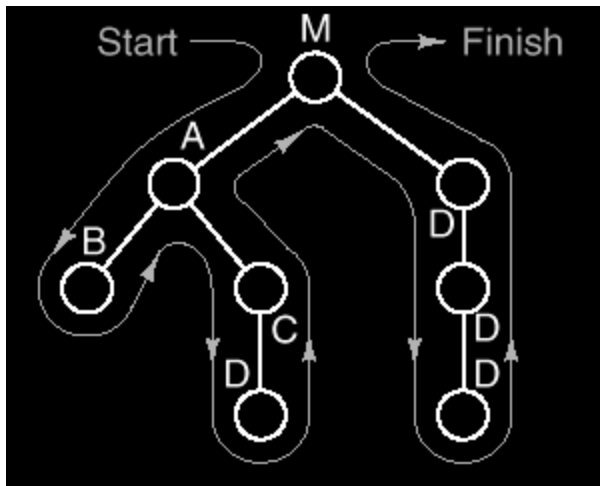
- Program Analysis and Program Design are closely interrelated. A good computer engineer must know both
- **Goal:** learn a host of new powerful programming techniques, plus formal methods for analyzing their performance.

# Example

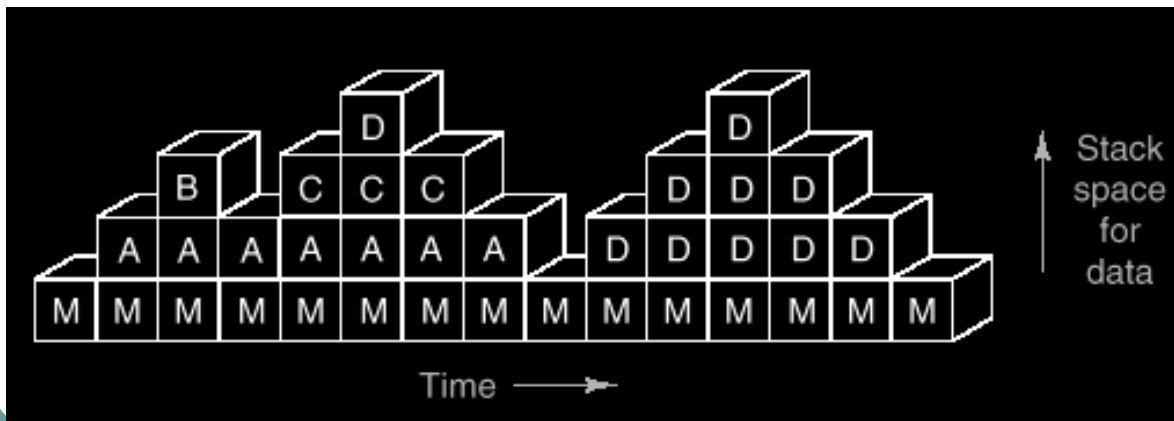
- Tree of Subprogram Calls



# Recursion - Introduction



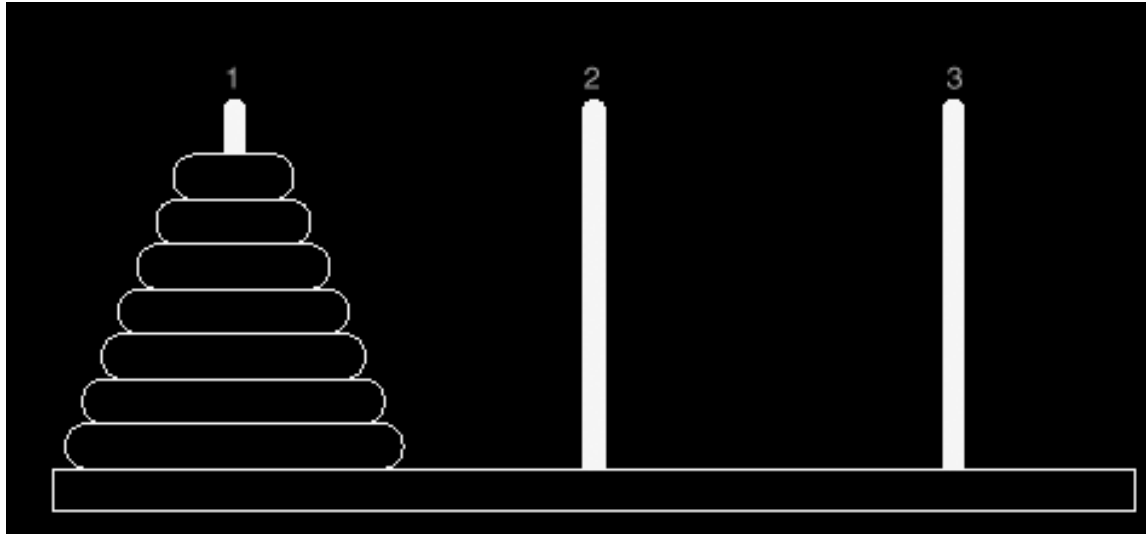
In recursive program instead of calling a different routine, one routine can repeatedly call itself.



# Definition & Use

- **Def:** Recursion is the case when a subprogram invokes itself or invokes a series of other subprograms that eventually invokes the first subprogram again.
- Recursion is a powerful tool to divide and conquer complex problems.
- It is important to carefully analyze a recursive solution

# Example 1: Tower of Hanoi



**This is task which is underway at the Temple of Brahma. At the creation of the world, the priest were given a brass platform on which were 3 diamond needles. On the first needle were stacked 64 golden disks, each one slightly smaller than the one under it. The priest were assigned the task of moving all the golden disks from the first needle to the third. The end of the task will signify the end of the world.**

# Rule

- You can not put a larger disk on top of the smaller one

# Question

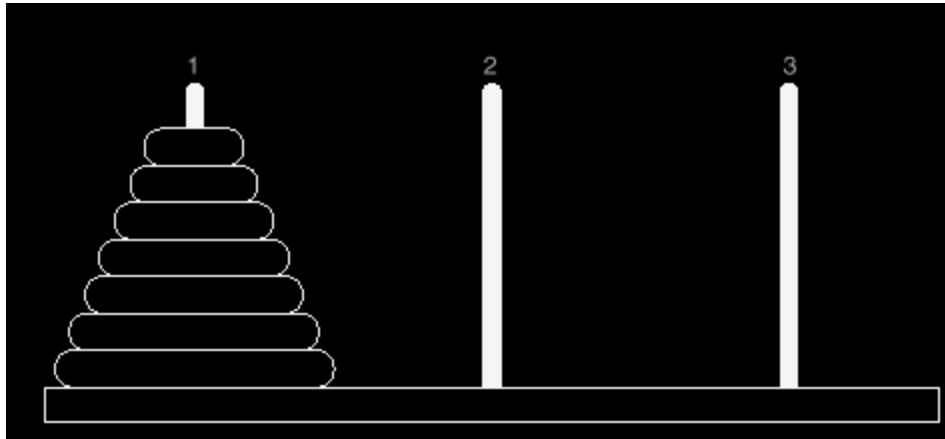
- How would you solve this problem?



# Solution

## ● Move(64, 1, 3, 2)

- Move 64 disks from tower 1 to tower 3 using tower 2 as temporary.

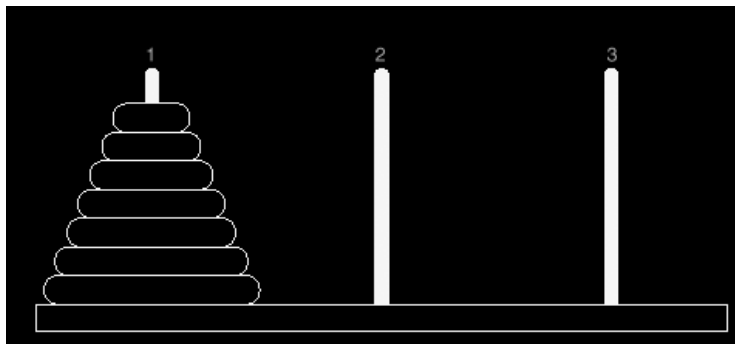


# Solution – Divide & Conquer

- Phase 1

- Move (63,1,2,3)
- `printf("Move disk #64 from tower 1 to tower 3\n");`
- Move(63,2,3,1)

- Phase 2 ... ?



- Phase 3 problem solved

# Movie

- <http://www.youtube.com/watch?v=9ISQ18s2EFI&fmt=18>

# Definition: Recursive Process

1. A smallest, base case that is processed without recursion
2. General method that reduces a particular case to one or more of the smaller cases, thereby making progress toward eventually reducing the problem all the way to the base case

# Designing a recursive function to solve the problem

```
int Move(int count, int start, int finish, int temp);
```

*Pre:* There are at least count disks on the tower start. The top disk (if any) on each of towers temp and finish is larger than any of the top count disks on tower start.

*Post:* The top count disks on start have been moved to finish; temp (used for temporary storage) has been returned to its starting position.

***/\* Move: moves count disks from start to finish using temp for temporary storage. \*/***

***void Move(int count, int start, int finish, int temp)***

***{***

***if (count > 0)***

***{***

***Move(count-1, start, temp, finish);***

***printf("Move a disk from %d to %d.\n", start, finish);***

***Move(count-1, temp, finish, start);***

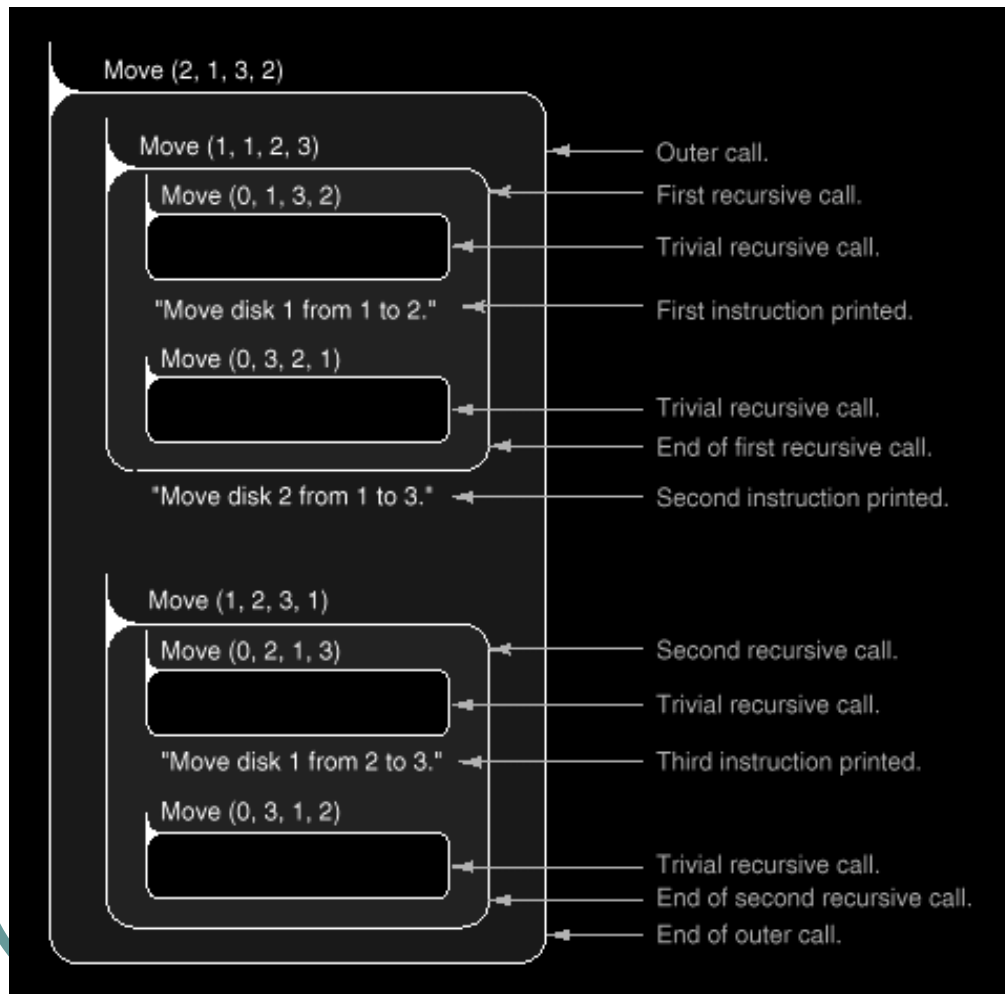
***}***

***}***

# Demo

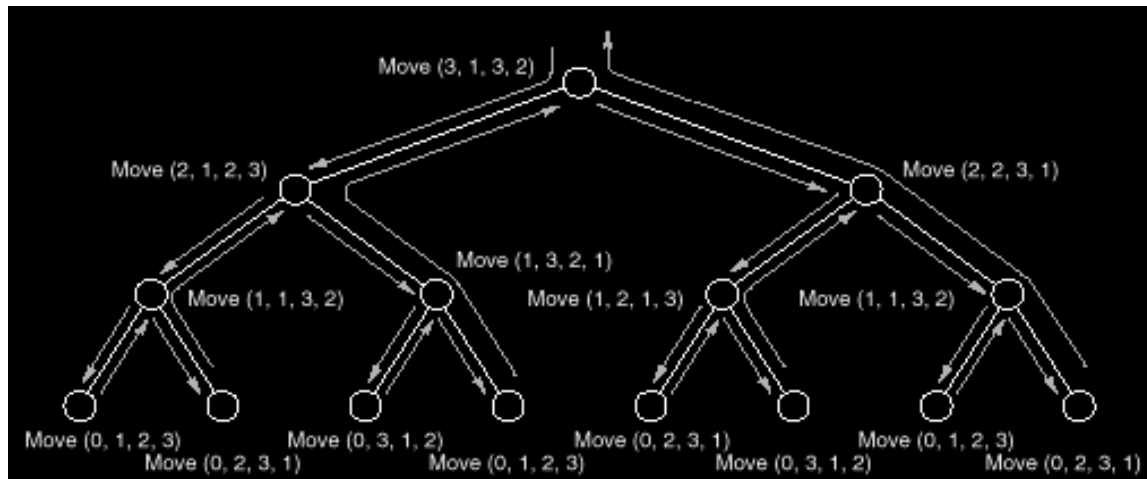
- <http://www.irt.org/games/js/hanoi/index.htm>

# Example: 2 disks version



# Analysis

- Recursion Tree



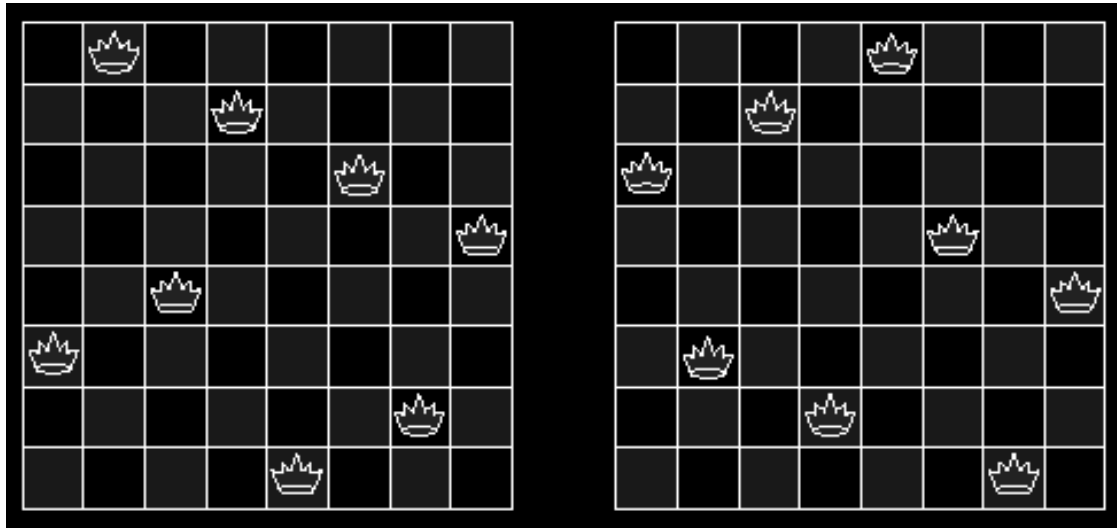
- Instructions are printed for each vertex of the tree except the leaves
- Number of nodes except for the leaves =  $1+2+4+\dots+2^{63}$   
 $= 2^{64} - 1$
- What is the running time measured in O notation?



# How large is the number of calls

- $2^{10} > 10^3$
- If priests can perform
  - one trivial operation per second (what is trivial operation here?)
  - $2^{64} > 2^4 \times 2^{60} = 16 \times 10^{18} \text{sec}$
- There are about
  - $3.2 \times 10^7$  seconds in a year
  - Our universe is ~ 20 billions years old
  - It will take 25 times more to complete the task
- How much space will be required?

# Example 2: 8-queen problem





**Apparently an analytically unsolvable problem. Even C. F. Gauss, who attempted this in 1850 was perplexed by this problem. But, solution do exists. See the two shown above.**




# Solution Idea

```
void
AddQueen(void)
{
    for (every unguarded position p on the
board) {
        Place a queen in position p;
        n++;
        if (n == 8)
            Print the configuration;
        else
            AddQueen();
        Remove the queen from position p;
        n--;
    }
}
```

# 4 queen problem

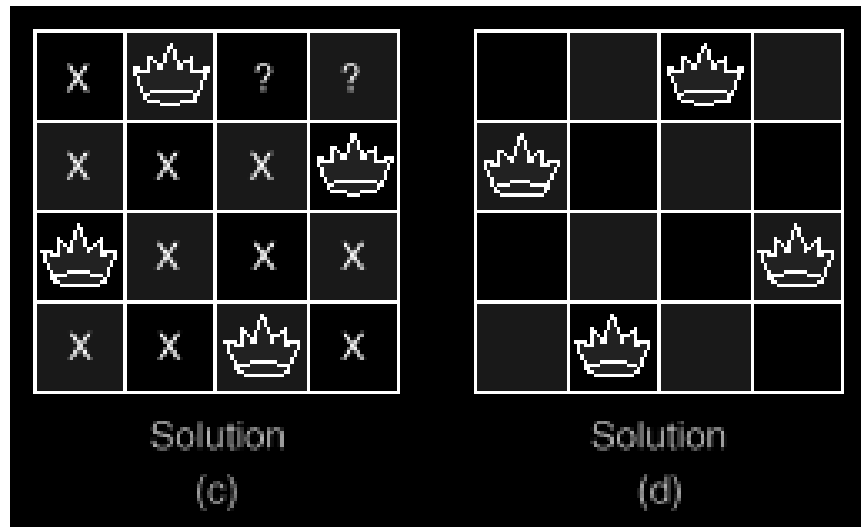
	?	?	?
X	X		?
X	X	X	X

Dead end  
(a)

	?	?	?
X	X	X	
X		X	X
X	X	X	X

Dead end  
(b)

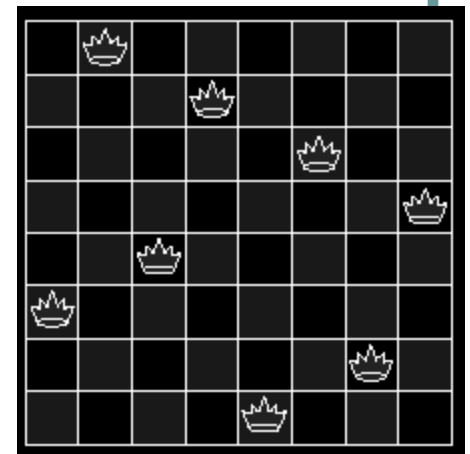
# 4 queen problem



**Backtracking: Build correct partial solution and proceed.  
When an inconsistent state arises, the algorithm backs up to  
the point of last correct partial solution.**

# Data Structure Choice

- Boolean or integer array?
  - Need help backtracking
- Pigeon hole principle
  - use one row, one queen to reduce the search
- Keep track of free columns
  - `int col[8]`
- Keep track of free diagonals
  - number of diagonal  $2 \times \text{boardsize} - 1$
  - all ~~down~~diagonal  $(x-y)=\text{constant}$
  - all ~~upward~~diagonal  $(x+y)=\text{constant}$



```
#include "common.h"
#define BOARD_SIZE 8
#define DIAGONAL (2*BOARD_SIZE-1)
#define DOWN_OFFSET 7
void WriteBoard(void);
void AddQueen(void);
int queencol[BOARD_SIZE]; /* column with the queen */
Boolean colfree[BOARD_SIZE]; /* Is the column free? */
Boolean upfree[DIAGONAL]; /* Is the upward diagonal free? */
Boolean downfree[DIAGONAL]; /* Is the downward diagonal free? */
int queencount = -1, /* row whose queen is currently placed
*/
numsol = 0; /* number of solutions found so far
*/

int main(void)
{
    int i;
    for (i = 0; i < BOARD_SIZE; i++)
        colfree[i] = TRUE;

    for (i = 0; i < DIAGONAL; i++)
    {
        upfree[i] = TRUE;
        downfree[i] = TRUE;
    }
    AddQueen();
    return 0;
}
```

```
void AddQueen(void)
{
    int col; /* column being tried for the queen */
    queencount++;
    for (col = 0; col < BOARDSIZE; col++)
        if (colfree[col] && upfree[queencount + col] &&
            downfree[queencount - col + DOWNOFFSET])
        {
            /* Put a queen in position (queencount, col). */
            queencol[queencount] = col;
            colfree[col] = FALSE;
            upfree[queencount + col] = FALSE;
            downfree[queencount - col + DOWNOFFSET] = FALSE;

            if (queencount == BOARDSIZE-1) /* termination condition*/
                WriteBoard();
            else
                AddQueen(); /* Proceed recursively.*/

            colfree[col] = TRUE; /* Now backtrack by removing the queen. */
            upfree[queencount + col] = TRUE;
            downfree[queencount - col + DOWNOFFSET] = TRUE;
        }
    queencount--;
}
```



# Analysis

- Naïve approach

- generate a random configuration and test it

$$\binom{64}{8} = 4,426,165,368$$

- One queen per row

- $8^8 = 16,777,216$

- One queen per column

- $8! = 40,320$

# Assignment

- Project 1