# EE 368

# Weeks 4 (Notes)

## Recursion and Backtracking

Read Chapter 3

**Recursion**

- Recursive Definition
- Some Examples
- Pros and Cons

A Class of Recursive Algorithms (steps or mechanics about performing some computation).

1.Divide and Conquer Algorithms

2.Backtracking

## A function invoking itself

```
void Countdown(int N)
{
  if (N >= 0)
  {
    printf("%d...\n", N);
    Countdown(N-1);
  }
  return;
}
int main()
{
  Countdown(10);
  return 0;
}
```

## Recursion vs Iteration

- When you write a function that invokes itself, the practice is called *recursion*. (the function recurs). Postponement of work.
- For many computations, there is a way to write it *recursively* and a way to write it *iteratively*.
- The iterative version is often more efficient.
- The recursive way is often more convenient.

## Iteration

- Iterative function: in which a computation is "unfolded" into smaller repeated steps. Explicit repetition of the same process until a certain condition is met.

- Example: iterative definition of factorial

```
prod = 1;
for ( x =n; x > 0; x--)
prod *=  x;
return(prod);
```

# Recursion

- Recursive function: computation is expressed in terms of a simpler version of itself.

- Example: Recursive definition of factorial

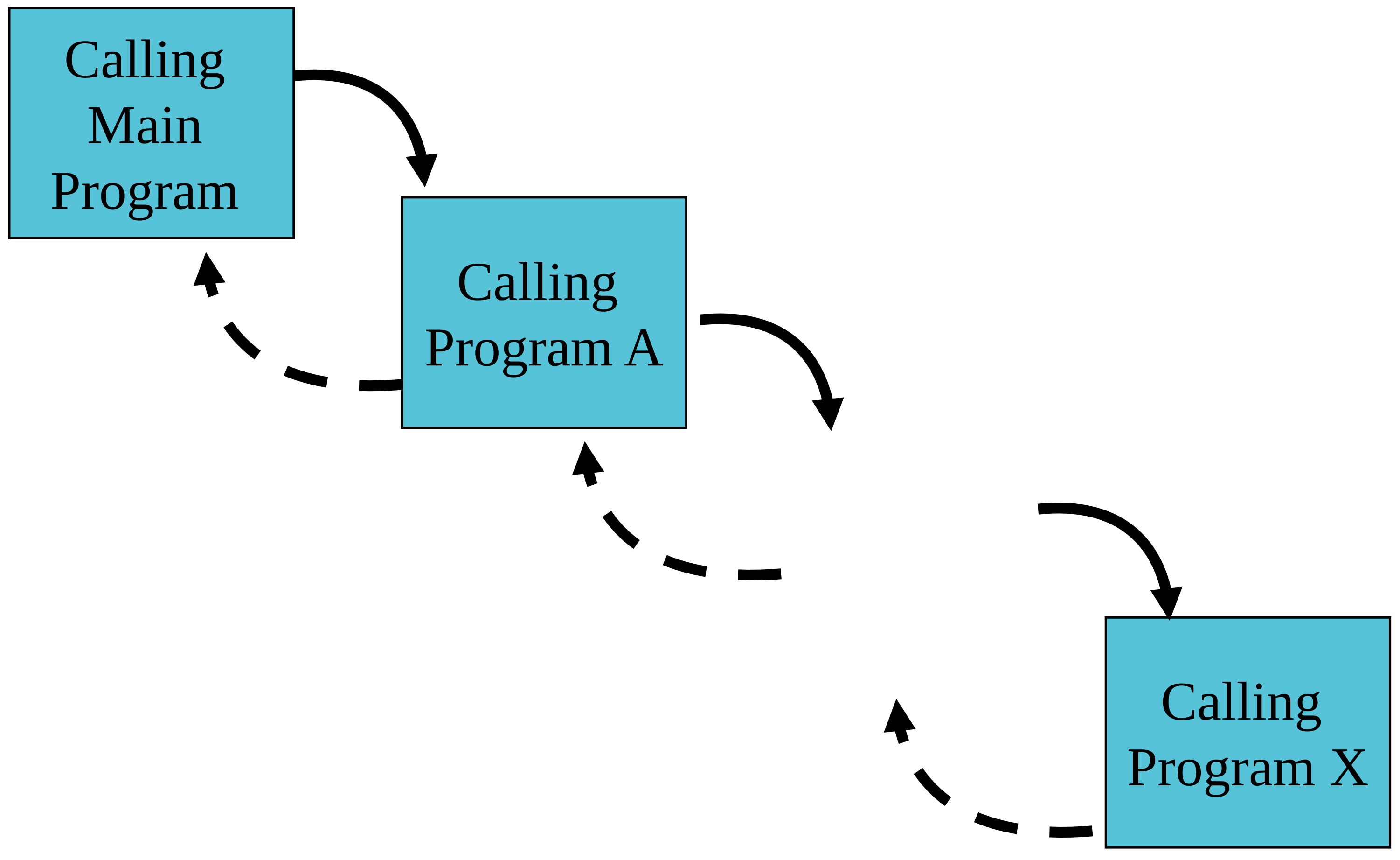
$$n! = \begin{cases} 1 & n == 0 \\ n * (n-1)! & n > 0 \end{cases}$$

# Factorial

```
int Factorial(int N)
{
  if (N == 0)
    return 1;

  return N * Factorial(N-1);
}
```

# How does recursion work



# Fibonacci sequence

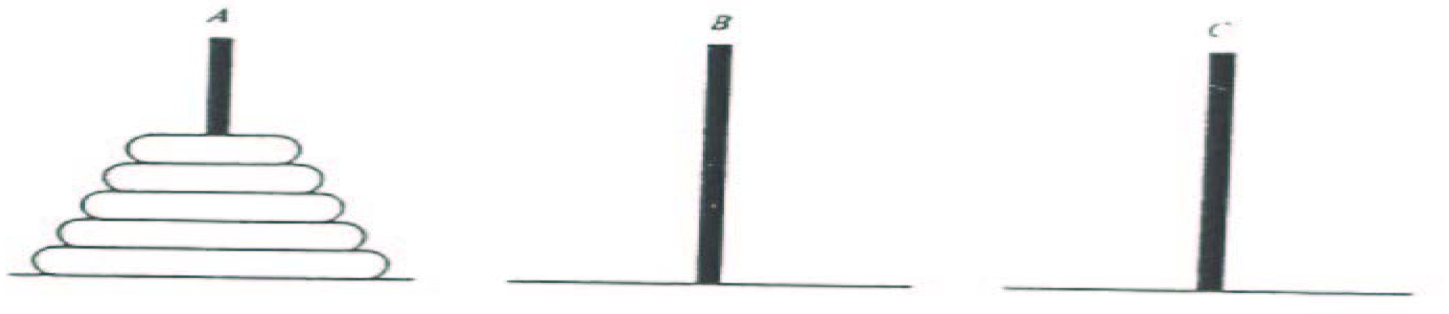
```
/* * Compute one number in the
 * Fibonacci sequence:
 * 1 1 2 3 5 8 13 21 34 55 89 144...
 */
int Fibonacci(int N)
{
  if (N == 0)
    return 1;
  if (N == 1)
    return 1;
  return ( Fibonacci(N-1) + Fibonacci(N-
 2) );
}
```

**It is Divide & Conquer algorithm**

## When using recursion...

- You always need to tell the function when to stop invoking itself.
- Don't return a pointer to something on the stack (i.e. don't return an address of a local variable)
- Don't perform recursion too deeply... you will run out of stack space.

# The Towers of Hanoi



**Object:** Move all Disks from peg A to C using peg B.

**Algorithm?**

**Number of Moves?**

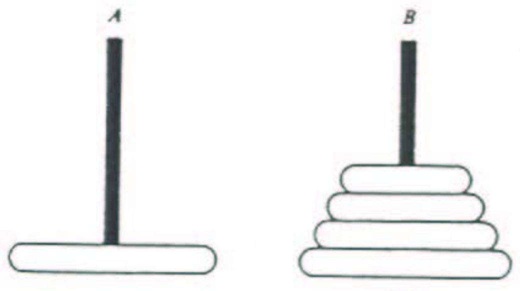
Note:

Only smaller disks can be placed on larger disks and disks can only be moved one at a time.

Way to solve recursively is to consider how to move bottom disk to peg C.

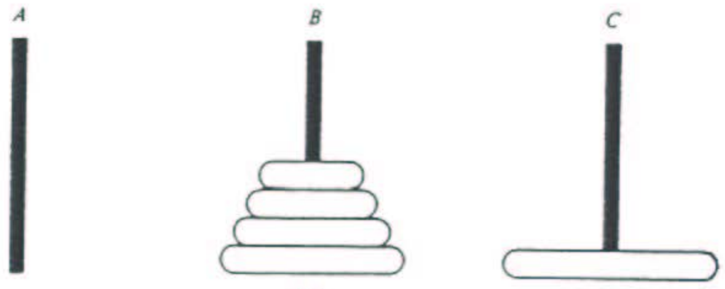
## Recursive Solution

Step 1: Move tower of 4 blocks to Peg B using Peg C.



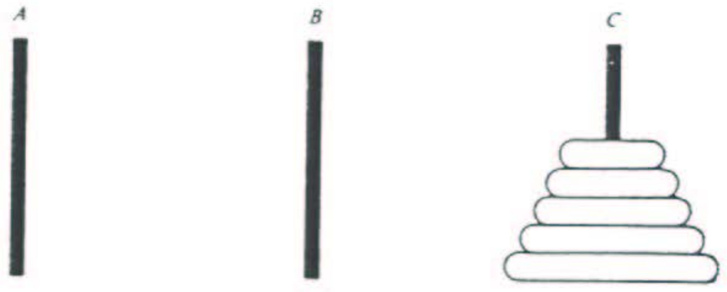
(a)

Step 2: Move largest disk from Peg A to Peg C.



(b)

Step 3: Move tower of 4 blocks from Peg B to Peg C using Peg A.



(c)

## General Recursive Solution

Consider case:  n disks & 3 pegs.

Pegs will be designed as **frompeg, topeg, auxpeg** depending on their role in our solution.

**Algorithm:**

IF n = 1, then move single disk from A to C

Else

1. Move n-1 disks from A to B using C as auxiliary peg
2. Move remaining disk from A to C.
3. Move n-1 disks from B to C using A as auxiliary disk.

# Implementation

```
# include  < stdio.h >
main ( )
{           int  n;
            scanf ( “%d”,  & n );
            towers ( n,  ‘A’,  ‘C’,  ‘B’ );
}            /* end main */
towers ( n,  frompeg,  topeg,  auxpeg )
int  n;
char auxpeg, frompeg, topeg;
{           /* If only one disk, make the move and return.                              */
            if ( n  ==  1 ) {
              printf ( “ \n%s%c%s%c”,  “move disk 1 from peg “,  frompeg,“ to peg “,  topeg);
                        return;
            } /* end if */
            /* Move top  n - 1  disks from  A  to  B, using  C  as                */
            */                                                                    auxiliary                           */
            towers ( n - 1,  frompeg,  auxpeg,  topeg );
            /*          move remaining disk from  A to  C                                           */
            printf ( “ \n%s%d%s%c%s%c”,  “move disk “,  n,  “ from peg “,
                                                                                            frompeg,  “ to peg “,  topeg );
            /*          Move  n - 1,  disk from  B  to  C  using  A  as             */
            /*                                                                    auxiliary                           */
            towers ( n - 1,  auxpeg,  topeg,  frompeg );
}            /* end towers */
```

**Note:  re-execution of the algorithm itself.**

## Recursion Tree

Tree exhibiting recursive calls. Can be used to find the total amount of computation (steps) of the algorithm

**Example: The Towers of Hanoi (to be done in class)**

## Solving Recurrence Equations

## (closed form expression for complexity)

1. Expanding Recurrences
2. Using General Solution

(homogeneous and particular solution, identical to solving partial differential equations)

# Example of Expanding Recurrence (Towers of Hanoi)

Number of moves needed to move a stack with n disks is given by:

$$\text{Moves } M(n) = \begin{cases} 2M(n-1) + 1 & n > 1 \\ 1 & n = 1 \end{cases}$$

| n | 1 | 2 | 3 | 4 | 5 | … |
|---|---|---|---|---|---|---|
| M (n) | 1 | 3 | 7 | 15 | 31 | … |

It looks like $M(n) = 2^n - 1$

Exponential number of moves

## Backtracking

Attempt to find a solution to a problem by constructing partial solutions which are consistent with the requirements of the problem. If an attempt to further extend the solution fails, backtrack and look for other possibilities.

Good for those types of problems, where many possibilities exist, but few survive for further test.

Searching for all or even one solution.

Technique is used to prune a recursion tree to a manageable size. Situations that are already checked out can be used to prevent the later investigation of fruitless efforts.

## Example: Eight-Queens Problem

Problem: Place eight queens on a chessboard so that no queen can attack any other queen.

Some configurations showing the solution

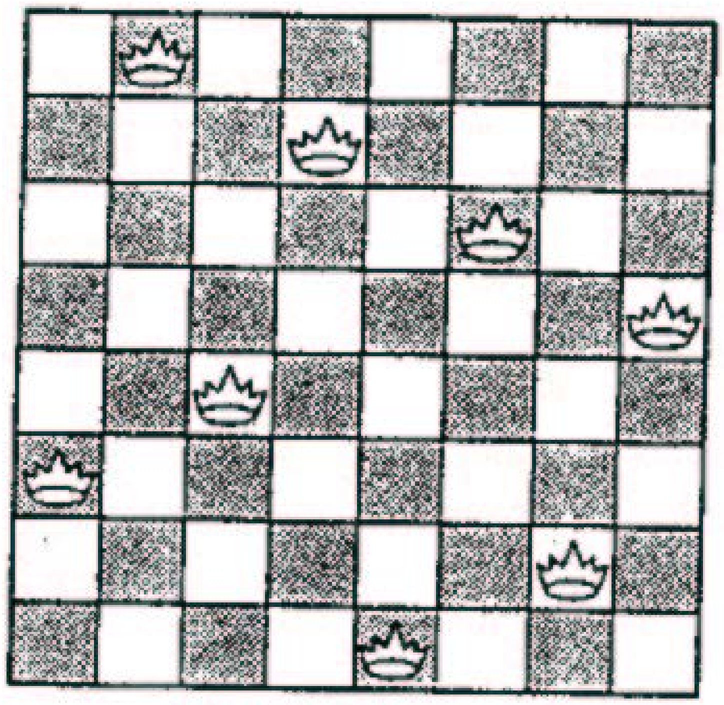
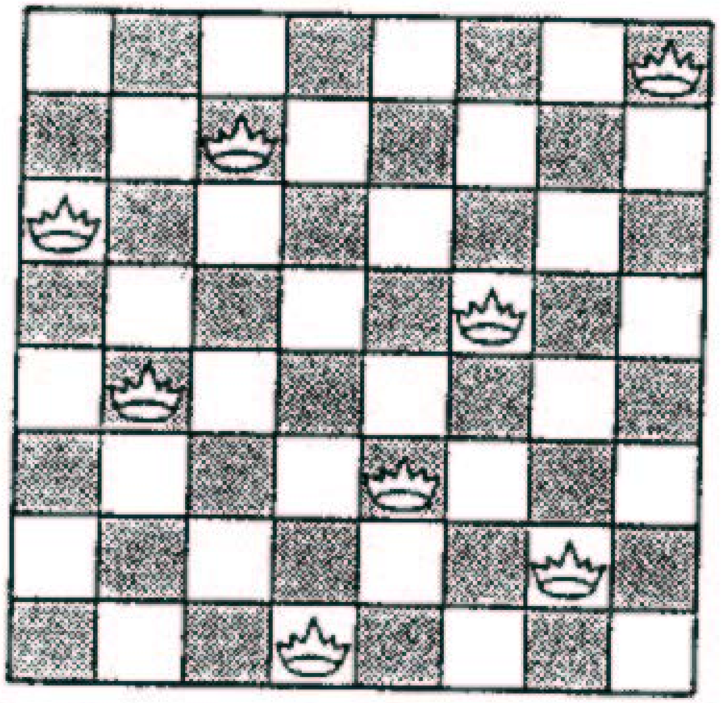



Figure 8.5. Two configurations showing eight nonattacking queens

## Search Procedure

Start with an exhaustive search on a 8 x 8 array.

Use Pigeonhole Principle to reduce search space:

Exactly one queen per row.

Also exactly one queen per column.

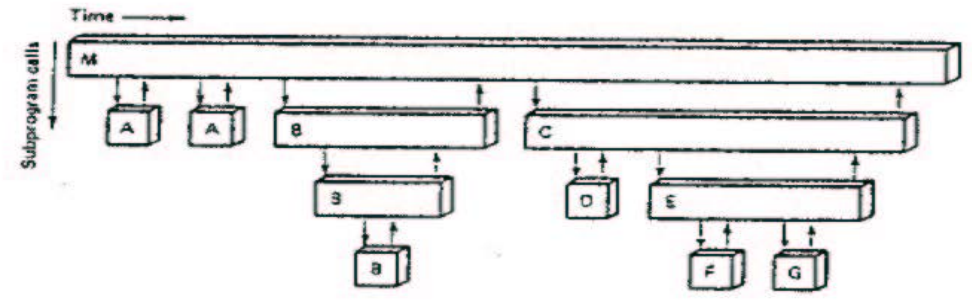
# An Implementation

```
static  short  int  board  [ 8 ] [ 8 ];
#define  TRUE  1
#define  FALSE  0
main ( )
{           int i,  j;
            for ( i = 0;  i < 8;  i ++ )
                        for ( j = 0;  j < 8;  j ++ )
                            board [ i ] [ j ]  =  FALSE;
            if (try ( 0 )  ==  TRUE )
                        drawboard ( );
} /* end main */
try (n)
int n;
}
int i;
for ( i = 0;  i < 8;  i ++ ) {
            board [ n ] [ i ]  =  TRUE;
            if ( n  ==  7  &&  good ( )  ==  TRUE )
                        return  ( TRUE );
            if ( n < 7  &&  good ( )  ==  TRUE  &&  try ( n + 1 )  == TRUE
                        return  ( TRUE );
                        board [ n ] [ i ]  =  FALSE
            } /* end for */
            return  (FALSE);
} /* end try */
```

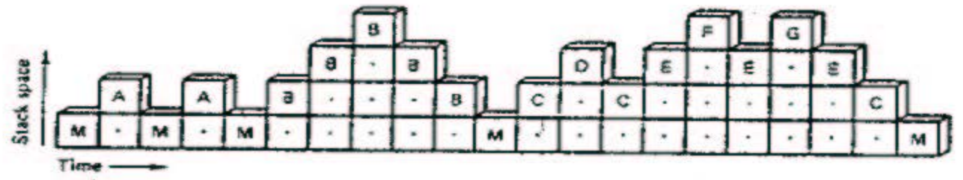
## Complexity and Improvements

## (to be done in class)

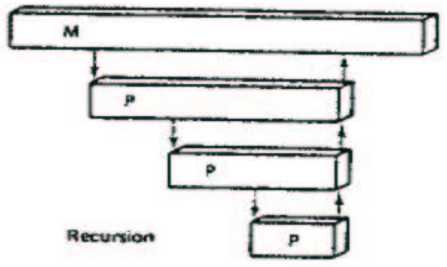
# Using Stack for Recursion



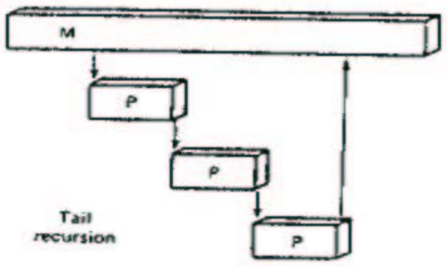
Time
Subprogram calls
M
A
A
B
C
B
D
E
B
F
G



Stack space
Time
M
A
B
C
D
E
F
G



M
P
P
P
Recursion



M
P
P
P
Tail recursion