Chapter 12 Recursion

- **recursion**: The definition of an operation in terms of itself.
  - Solving a problem using recursion depends on solving smaller occurrences of the same problem.

- **recursive programming**: Writing methods that call themselves to solve problems recursively.
  - An equally powerful substitute for *iteration* (loops)
  - Particularly well-suited to solving certain types of problems

n! (n factorial)

n! = 1 * 2 * 3 * … * n

- Compute n! Here's the obvious solution:

```
int fact = 1;
for (int i = 1; i <= n; i++) {
    fact *= i;
}
```

Make it a function

- Same thing as a function:

```
int factorial(int n) {
    int result = 1;
    for (i = 1; i <= n; i++) {
        result *= i;
    }
    return result;
}
```

A little Math

factorial(4)= 1 * 2 * 3 * 4

= (1 * 2 * 3) * 4

= fact(3) * 4

= (fact(2) * 3) * 4

= ((fact(1) * 2) * 3) * 4

- In general

factorial(n) = factorial(n – 1) * n

Let’s simplify our function

```
int fact(int n) {
    return fact(n - 1) * n;
}
```

- What happens if we try to compute fact(4)?

Let’s simplify our function

```
int fact(int n) {
    return fact(n - 1) * n;
}
```

- What happens if we try to compute fact(4)?
- Answer: Program fails: StackOverflowError

More precise definition

```
factorial(0)  = 1
factorial(n) = factorial(n-1) * n    if n > 0
```

- Now

```
factorial(4)    = factorial(3) * 4
                = (factorial(2) * 3) * 4
                = (factorial(1) * 2) * 3) * 4
                = (factorial(0) * 1) * 2) * 3) * 4
                = (1 * 1) * 2) * 3) * 4
```

Rewriting our function

```
int fact(int n) {
    if (n == 0)
        return 1;
    else
        return fact(n - 1) * n;
}
```

- Now if we compute fact(4) we get 24.

Some terminology

- n == 0 is a (or the) *base case*
  - We can compute factorial(0) directly
- n > 0 is the *recursive case* or *recursive step*
  - This case is computed by computing factorial for a smaller n. The program *recurs* or calls itself to compute the answer.

Some more exercises

1. Write a recursive mathematical definition for computing 2^n for a positive integer n.

2. Write a recursive mathematical definition for computing 1 + 2 + 3 + … + n for a positive integer.

Characteristics of Recursion

All recursive methods work the same basic way:

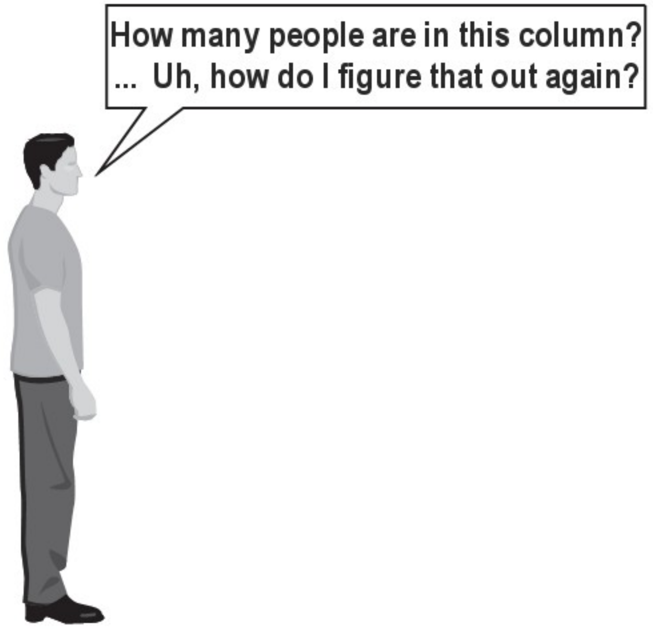
- There are one or more base cases where the answer is computed without recursion.
  - These cases stop the recursion.
- There are one or more recursive cases which compute a simpler version of the original problem.
  - These cases bring the solution closer to a base case.

Recursive Thinking: The General Approach

```
if problem is “small enough”
    solve it directly
else
    1. break into one or more smaller
       subproblems
    2. solve each subproblem recursively
    3. combine results into solution to the
       whole problem
```

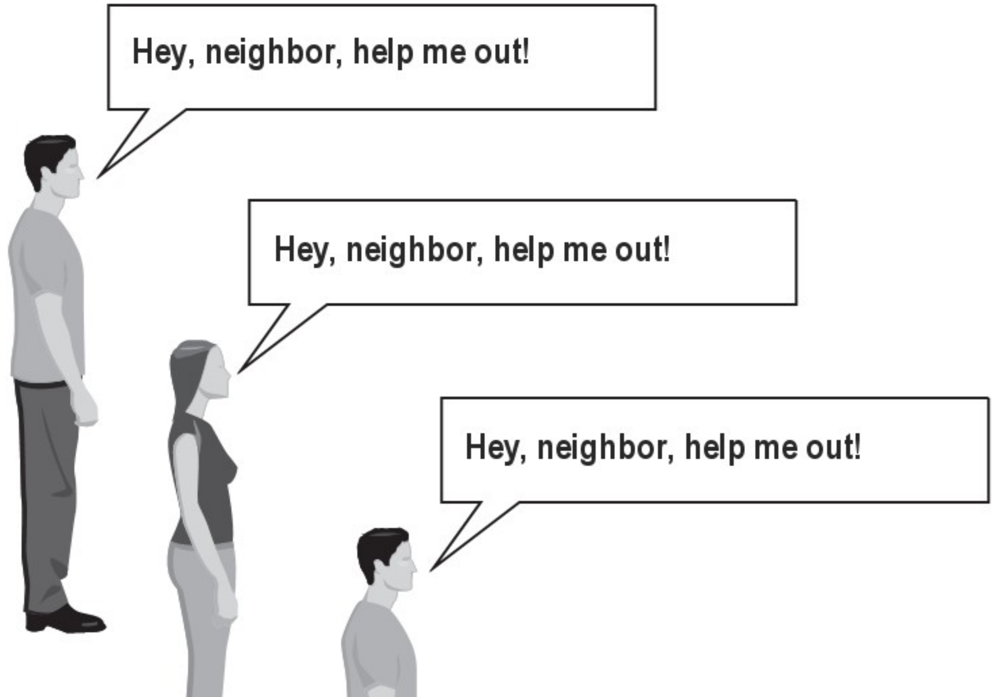
Another exercise

- (To a student in the front row)
  How many students total are directly behind you in your "column" of the classroom?
  - You have poor vision, so you can see only the people right next to you. So you can't just look back and count.
  - But you are allowed to ask questions of the person next to you.
  - How can we solve this problem? (*recursively*)


How many people are in this column?
... Uh, how do I figure that out again?

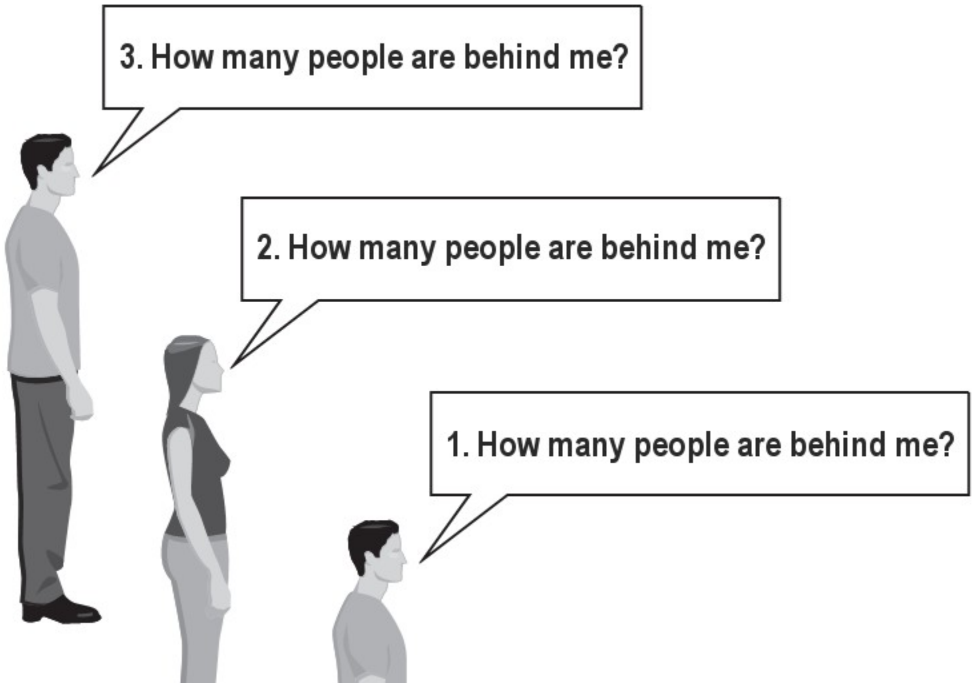
The idea

- Recursion is all about breaking a big problem into smaller occurrences of that same problem.
  - Each person can solve a small part of the problem.
    - What is a small version of the problem that would be easy to answer?
    - What information from a neighbor might help me?


Hey, neighbor, help me out!
Hey, neighbor, help me out!
Hey, neighbor, help me out!

Recursive algorithm

- Number of people behind me:
  - If there is someone behind me,
    ask him/her how many people are behind him/her.
    - When they respond with a value **N**, then I will answer **N + 1**.
  - If there is nobody behind me, I will answer **0**.


3. How many people are behind me?
2. How many people are behind me?
1. How many people are behind me?

Problem Solving Using Recursion

- Consider the following method to print a line of `*` characters:

```
// Prints a line containing the given number of stars.
// Precondition: n >= 0
public static void printStars(int n) {
    for (int i = 0; i < n; i++) {
        System.out.print("*");
    }
    System.out.println();   // end the line of output
}
```

- Write a recursive version of this method (that calls itself).
  - Solve the problem <u>without using any loops</u>.

A basic case

- What are the cases to consider?
  - What is a very easy number of stars to print without a loop?

```
public static void printStars(int n) {
    if (n == 1) {
        // base case; just print one star
        System.out.println("*");
    } else {
        ...
    }
}
```

Using recursion properly

- Condensing the recursive cases into a single case:

```
public static void printStars(int n) {
    if (n == 1) {
        // base case; just print one star
        System.out.println("*");
    } else {
        // recursive case; print one more star
        System.out.print("*");
        printStars(n - 1);
    }
}
```

"Recursion Zen"

- The real, even simpler, base case is an `n` of 0, not 1:

```
public static void printStars(int n) {
    if (n == 0) {
        // base case; just end the line of output
        System.out.println();
    } else {
        // recursive case; print one more star
        System.out.print("*");
        printStars(n - 1);
    }
}
```

  – **Recursion Zen**: The art of properly identifying the best set of cases for a recursive algorithm and expressing them elegantly.

Exercise

- Write a recursive method `isPalindrome` accepts a `String` and returns `true` if it reads the same forwards as backwards.
  - `isPalindrome("madam")` → `true`
  - `isPalindrome("racecar")` → `true`
  - `isPalindrome("step on no pets")` → `true`
  - `isPalindrome("able was I ere I saw elba")` → `true`
  - `isPalindrome("Java")` → `false`
  - `isPalindrome("rotater")` → `false`
  - `isPalindrome("byebye")` → `false`
  - `isPalindrome("notion")` → `false`

Exercise solution

```
// Returns true if the given string reads the same
// forwards as backwards.
// Trivially true for empty or 1-letter strings.
public static boolean isPalindrome(String s) {
    if (s.length() < 2) {
        return true;    // base case
    } else {
        char first = s.charAt(0);
        char last  = s.charAt(s.length() - 1);
        if (first != last) {
            return false;
        }               // recursive case
        String middle = s.substring(1, s.length() - 1);
        return isPalindrome(middle);
    }
}
```

Exercise solution 2

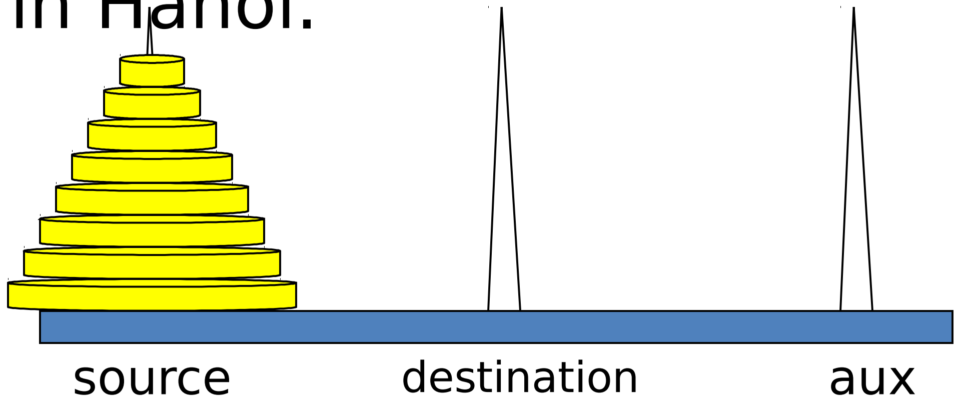
```java
// Returns true if the given string reads the same
// forwards as backwards.
// Trivially true for empty or 1-letter strings.
public static boolean isPalindrome(String s) {
    if (s.length() < 2) {
        return true;   // base case
    } else {
        return s.charAt(0) == s.charAt(s.length() - 1) &&
            isPalindrome(s.substring(1, s.length() - 1));
    }
}
```

More examples with recursion

- Towers of Hanoi
- Fibonacci Numbers

A Legend

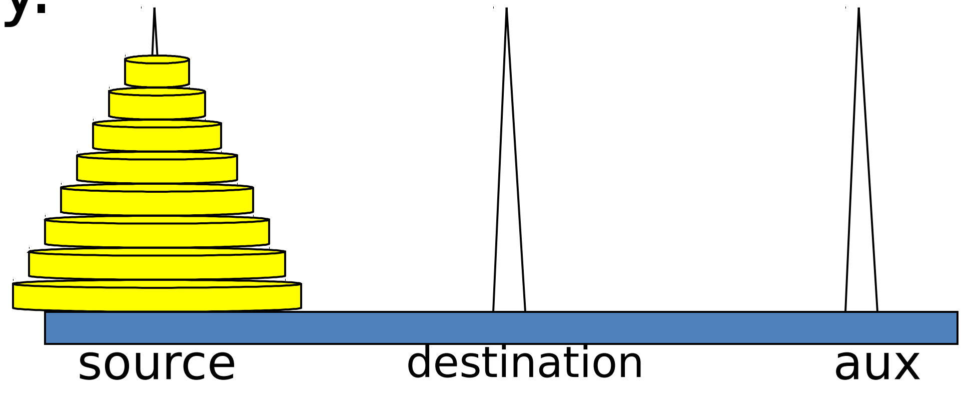
Legend has it that there were three diamond needles set into the floor of the temple of Brahma in Hanoi.



Stacked upon the leftmost needle were 64 golden disks, each a different size, stacked in concentric order:

A Legend (*Ct'd*)

The priests were to transfer the disks from the first needle to the second needle, using the third as necessary.

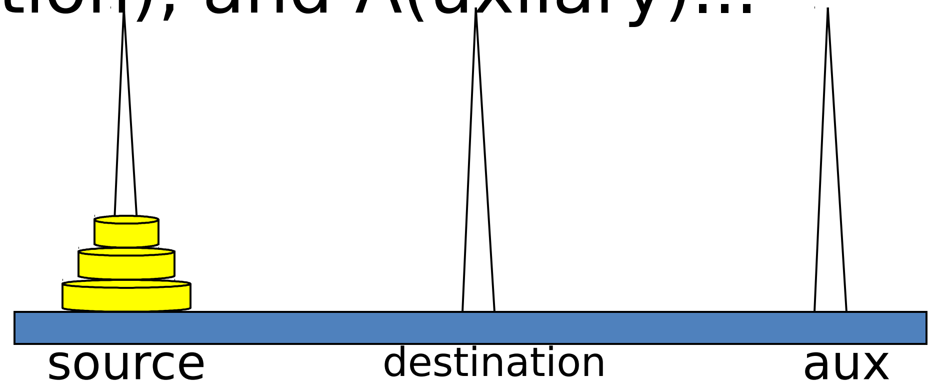


But they could *only move one disk at a time*, and could *never put a larger disk on top of a smaller one*.

When they completed this task, <u>the world would</u> end!

To Illustrate

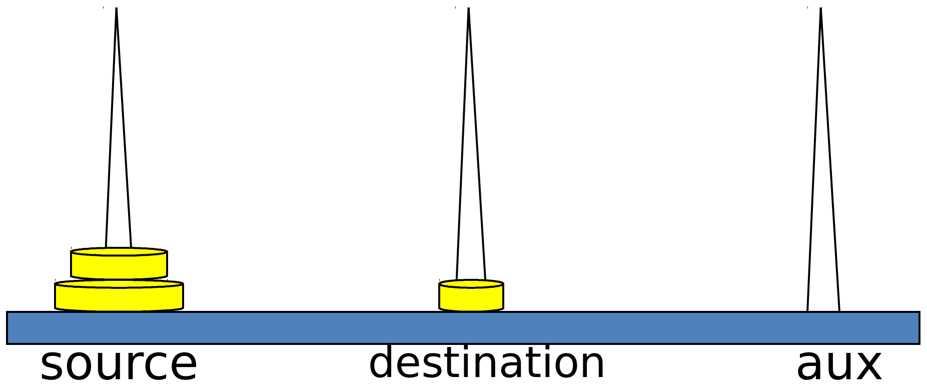
For simplicity, suppose there were just 3 disks, and we'll refer to the three needles as S(ource), D(estination), and A(uxilary)...



Since we can only move one disk at a time, we move the top disk from S to D.

Example

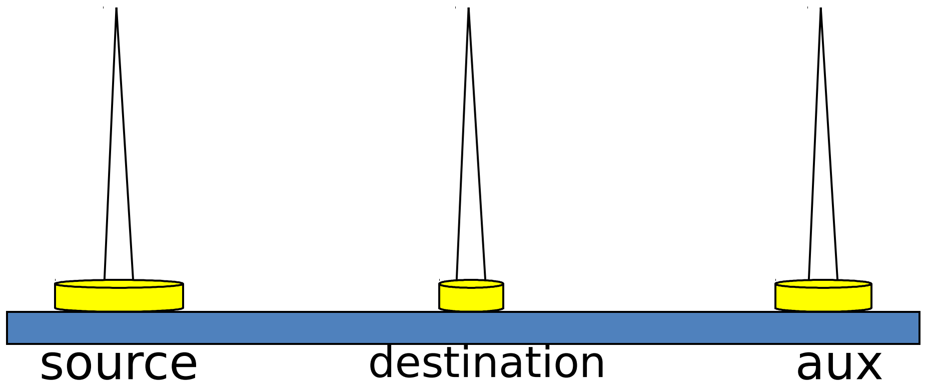
For simplicity, suppose there were just 3 disks, and we’ll refer to the three needles as S, D, and A...



We then move the top disk from S to A.

Example (*Ct'd*)

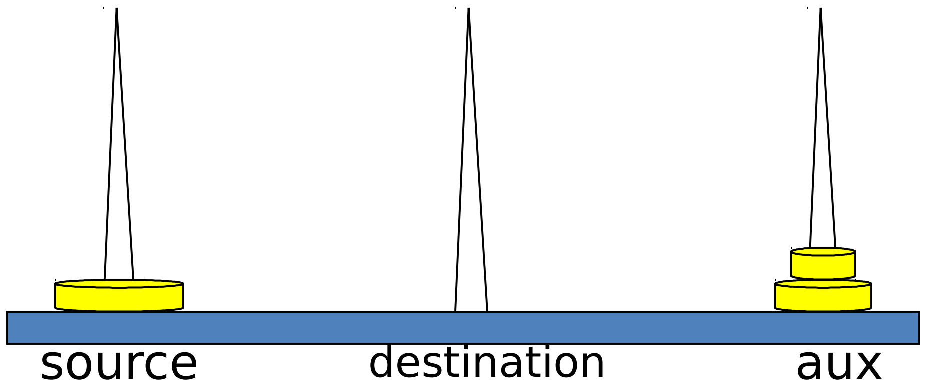
For simplicity, suppose there were just 3 disks, and we'll refer to the three needles as S, D, and A...



We then move the top disk from D to A.

Example (*Ct'd*)

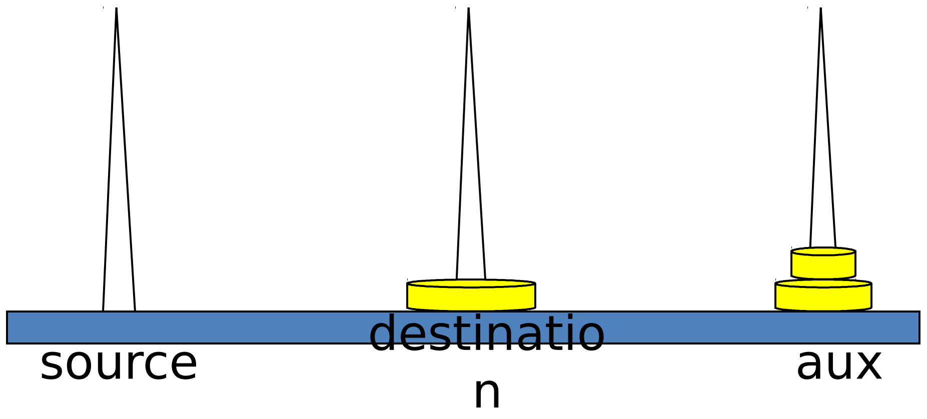
For simplicity, suppose there were just 3 disks, and we'll refer to the three needles as S, D, and A...



We then move the top disk from S to D.

Example (*Ct'd*)

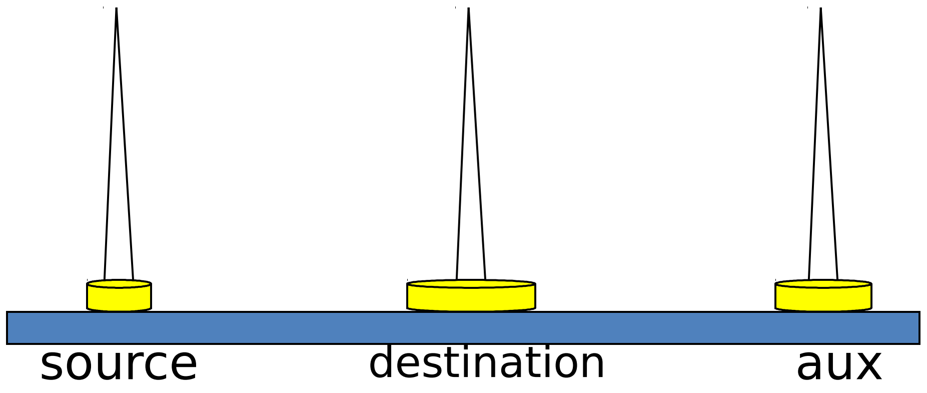
For simplicity, suppose there were just 3 disks, and we'll refer to the three needles as S, D, and A...



We then move the top disk from A to S.

Example (*Ct'd*)

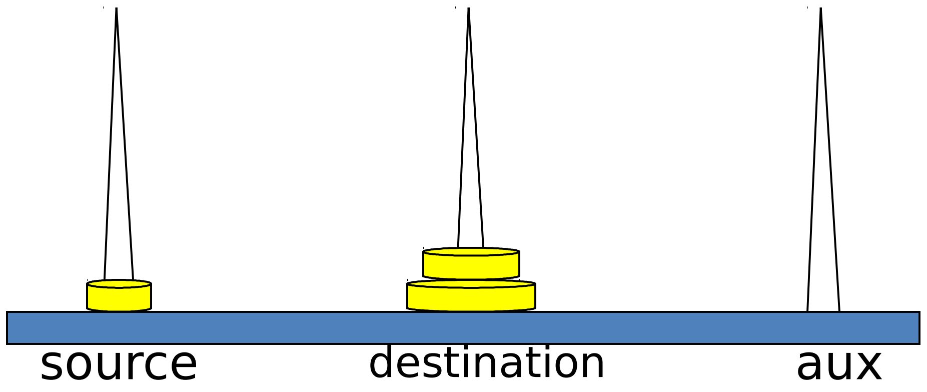
For simplicity, suppose there were just 3 disks, and we'll refer to the three needles as S, D, and A...



We then move the top disk from A to D.

Example (*Ct'd*)

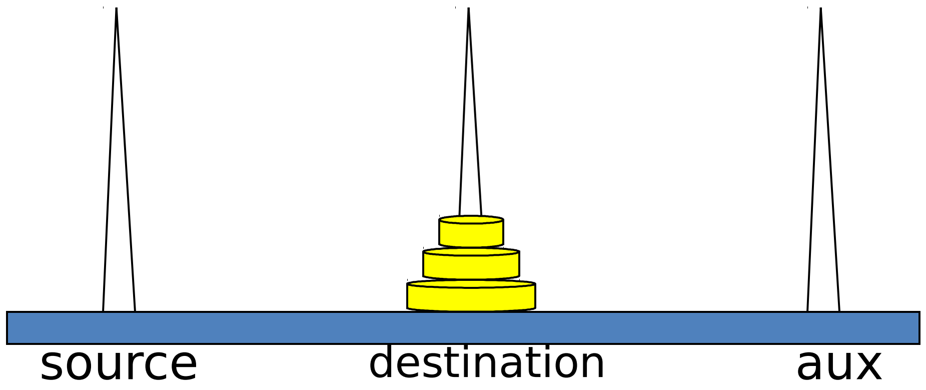
For simplicity, suppose there were just 3 disks, and we'll refer to the three needles as S, D, and A...



We then move the top disk from S to D.

Example (*Ct'd*)

For simplicity, suppose there were just 3 disks, and we'll refer to the three needles as S, D, and A...

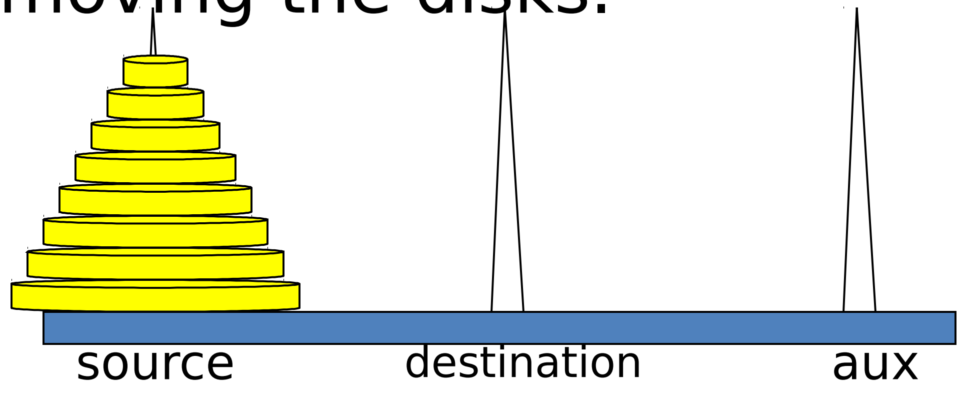


and we're done!

The problem gets more difficult as the number of disks increases...

Our Problem

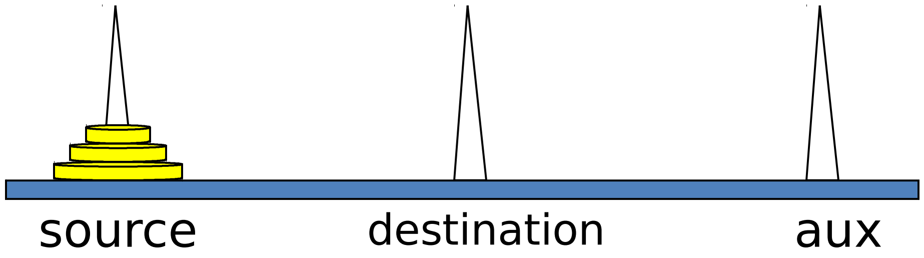
Today's problem is to write a program that generates the instructions for the priests to follow in moving the disks.



While difficult to solve iteratively, this problem has a simple and elegant *recursive* solution.

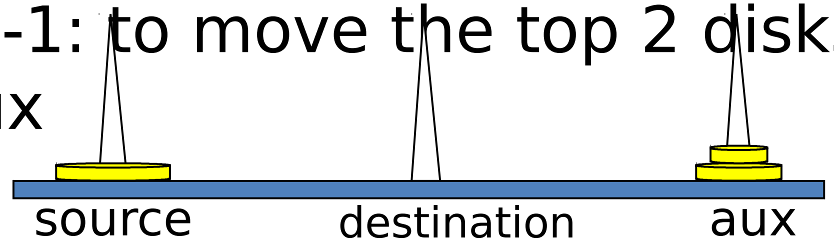
A closer look

Original problem:

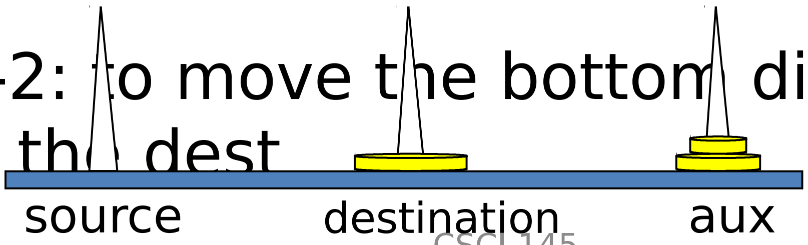


First goal: to move the bottom disk from the source needle to the destination needle. To achieve this goal:

sub-goal-1: to move the top 2 disks from the source to the aux

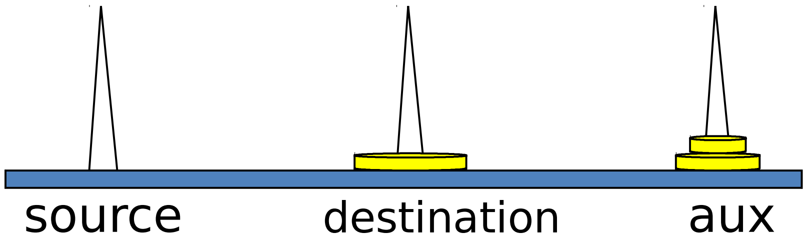


sub-goal-2: to move the bottom disk from the source to the dest



A closer look

Now the problem:



Second goal: move the 2 disks from the aux to the dest.

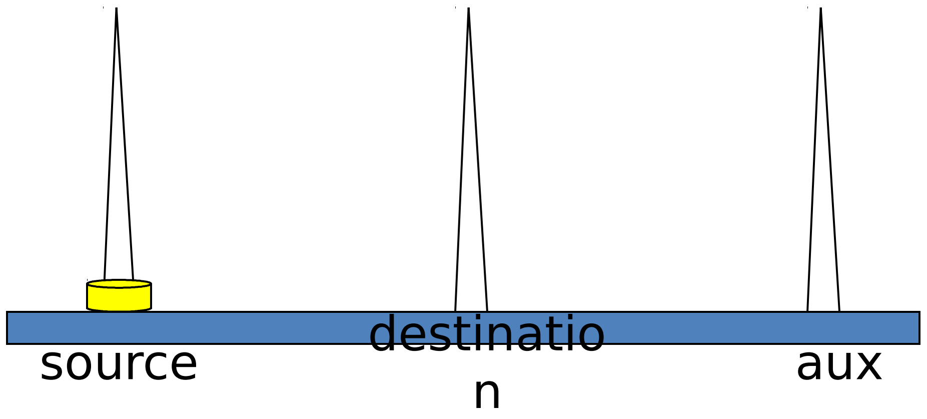
A similar problem is encountered if we take the aux as the source, the source as the aux.

Recursion comes in!

Recursive Strategy Design

Base case: What is an instance of the problem that is trivial?

→ n = 1

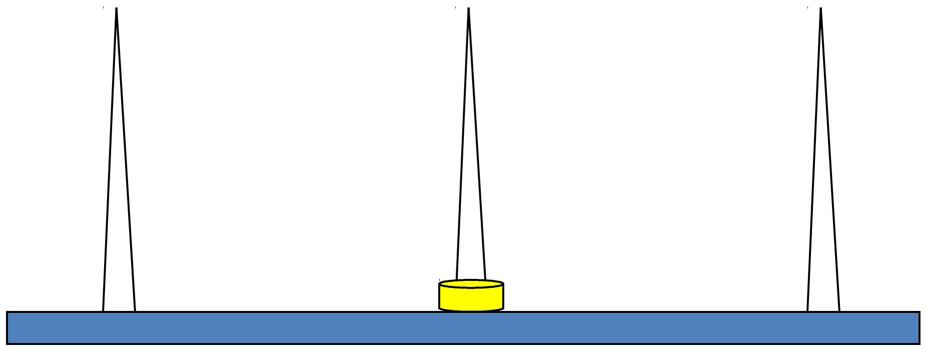


Since this base case could occur when the disk is on any needle, we simply output the instruction to move the top disk from *src* to *dest*.

Recursive Strategy Design

Base case: What is an instance of the problem that is trivial?

$\rightarrow$ n =1

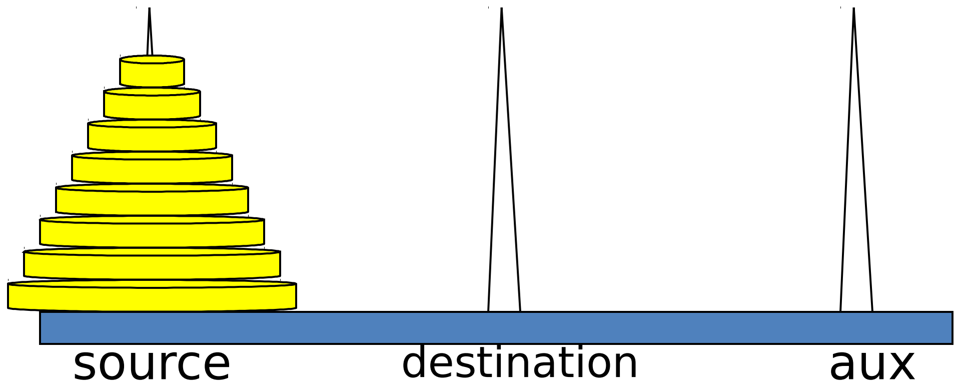


Since this base case could occur when the disk is on any needle, we simply output the instruction to move the top disk from *src* to *dest*.

Recursive Strategy Design

Recursive Step: n > 1

→ How can recursion help us out?

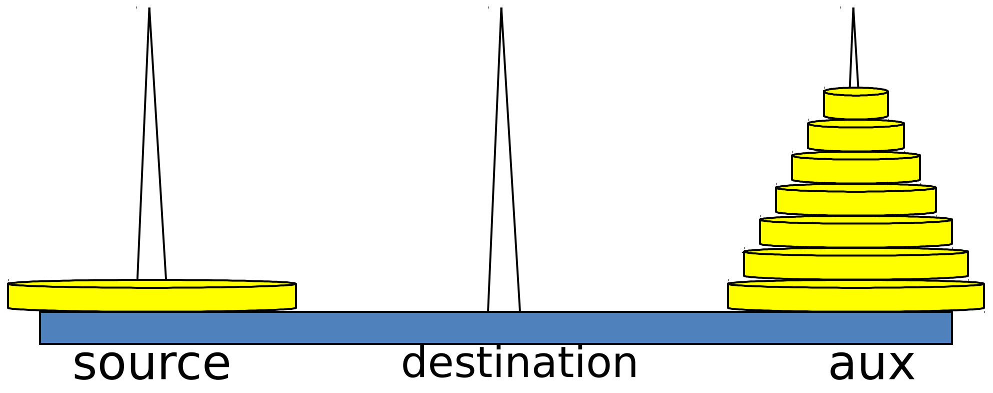


a. <u>Recursively</u> move n-1 disks from *src* to *aux*.

Recursive Strategy Design

Recursive Step: n > 1

→ How can recursion help us out?

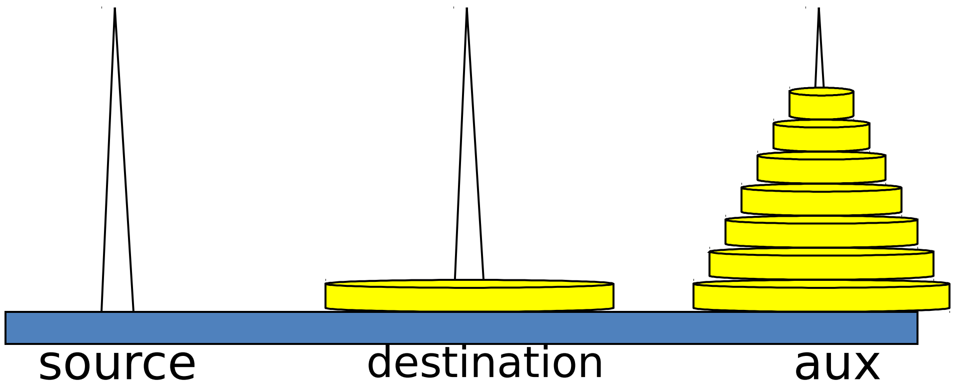


b. Move the one remaining disk from *src* to *dest*.

Recursive Strategy Design

Recursive Step: n > 1

→ How can recursion help us out?

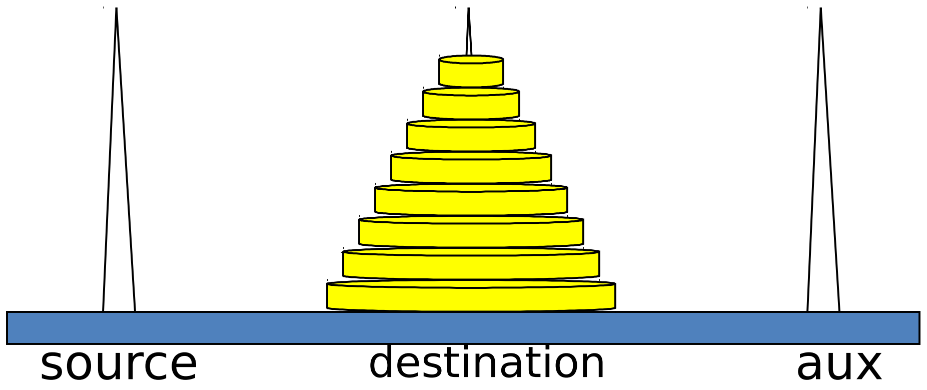


c. *Recursively* move n-1 disks from *aux* to *dest*...

Recursive Strategy Design

Recursive Step: n > 1

→ How can recursion help us out?

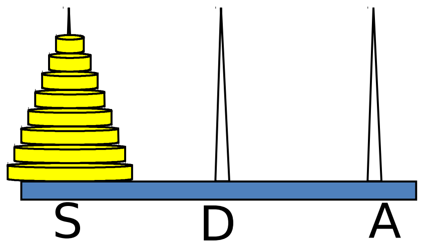


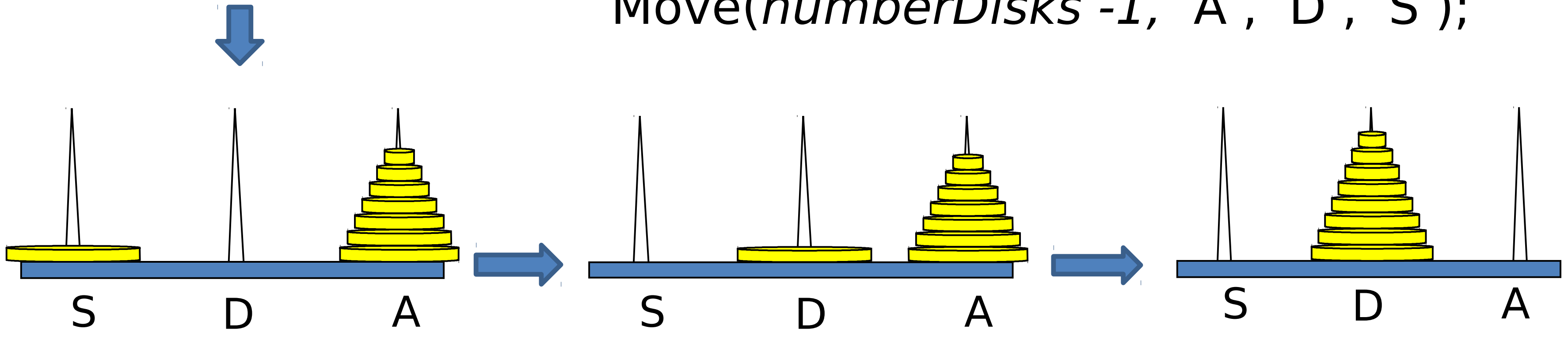
d. We’re done!

Algorithm

We can combine these steps into the following algorithm:

```
1. If numberDisks ==  1:
          Display “Move the top disk from  S to D”.
2.  Else
       Move(numberDisks -1, ‘S’, ‘A’, ‘D’);
       Move(1, ‘S’, ‘D’, ‘A’);
       Move(numberDisks -1, ‘A’, ‘D’, ‘S’);
```





More examples with recursion

- Towers of Hanoi
- Fibonacci Numbers

Fibonacci Numbers

```
Fibonacci series: 0 1 1 2 3 5 8 13 21 34 55 89…
         indices: 0 1 2 3 4 5 6  7  8  9 10 11
```

fib(0) = 0;

fib(1) = 1;

fib(index) = fib(index -1) + fib(index -2); index >=2

fib(3) = fib(2) + fib(1) = (fib(1) + fib(0)) + fib(1) = (1 + 0) +fib(1) = 1 + fib(1) = 1 + 1 = 2

How to code it?

Computing Fibonacci Numbers

- Fibonacci numbers are defined recursively:

  fib(0) = 0;

  fib(1) = 1;

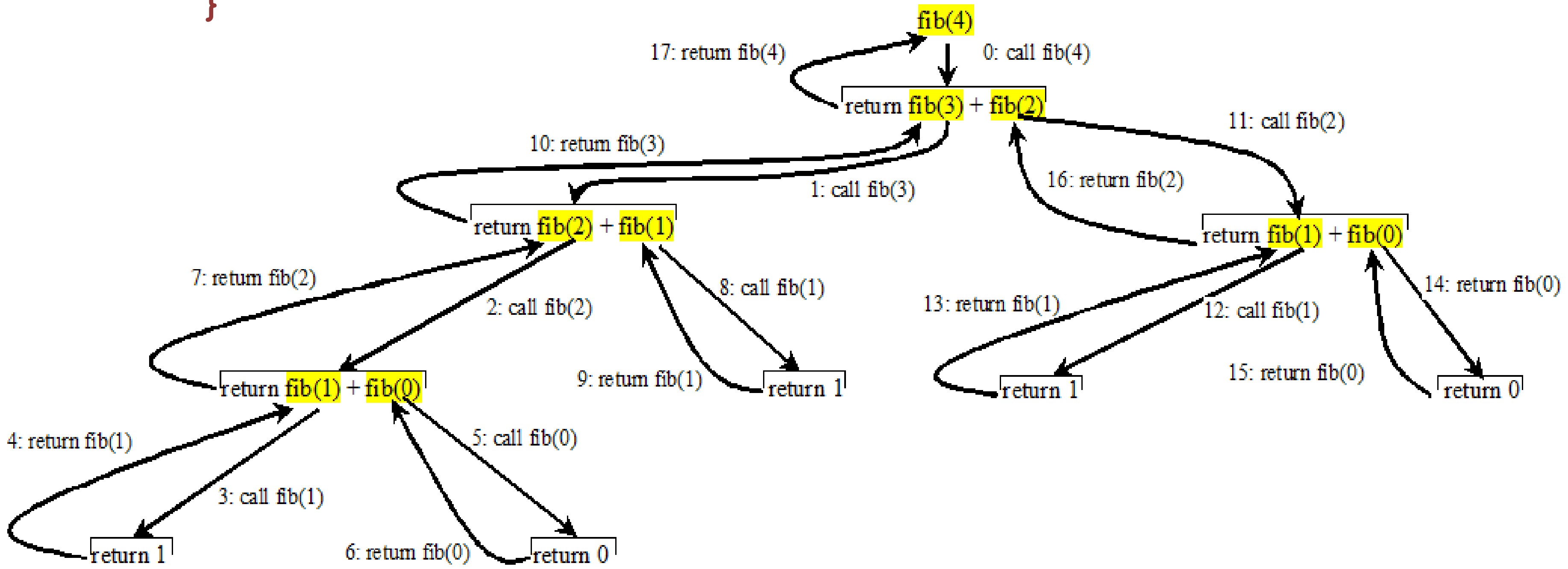
  fib(index) = fib(index -1) + fib(index -2); index >=2

- As a recursive algorithm (first attempt):

```
Algorithm Fib($k$):
    Input: non-negative $k$
    Output: The $k$th Fibonacci number $F_k$
  if $k$ ℜℑ 1 then
    return $k$
  else
    return Fib($k$ ↖ 1) + Fib($k$ ↖ 2)
```

Fibonnaci Numbers, cont.

```
public static long fib(int n) {
  if (n <= 1)
    return n;
  else
    return fib(n-1) + fib(n-2);
}
```



This is straightforward, but an inefficient recursion ...

Efficient Fibonacci

- ***Strategy:*** keep track of:
  - *Current* Fibonacci number
  - *Previous* Fibonacci number

Efficient Fibonacci in Python

```
def fib(n):
    if n == 0:
        return 0
    else:
        return fibaux(n)[0]

def fibaux(n):
    if n == 1:
        return 1, 0
    else:
        f2, f1 = fibaux(n – 1)
 return f2 + f1, f2
```

Primary function

Special case handled here

Call auxiliary function to do real work and return first of two returned values.

Auxiliary function does the recursion.

Auxiliary function returns two values: fib(n) and fib(n-1)

Recursion: How it works

Main:

```
    fib(3)
```

Recursion: How it works

Main:

fib:

```
    n = 3 // from call to fib
    return fibaux(3)[0]
```

Recursion: How it works

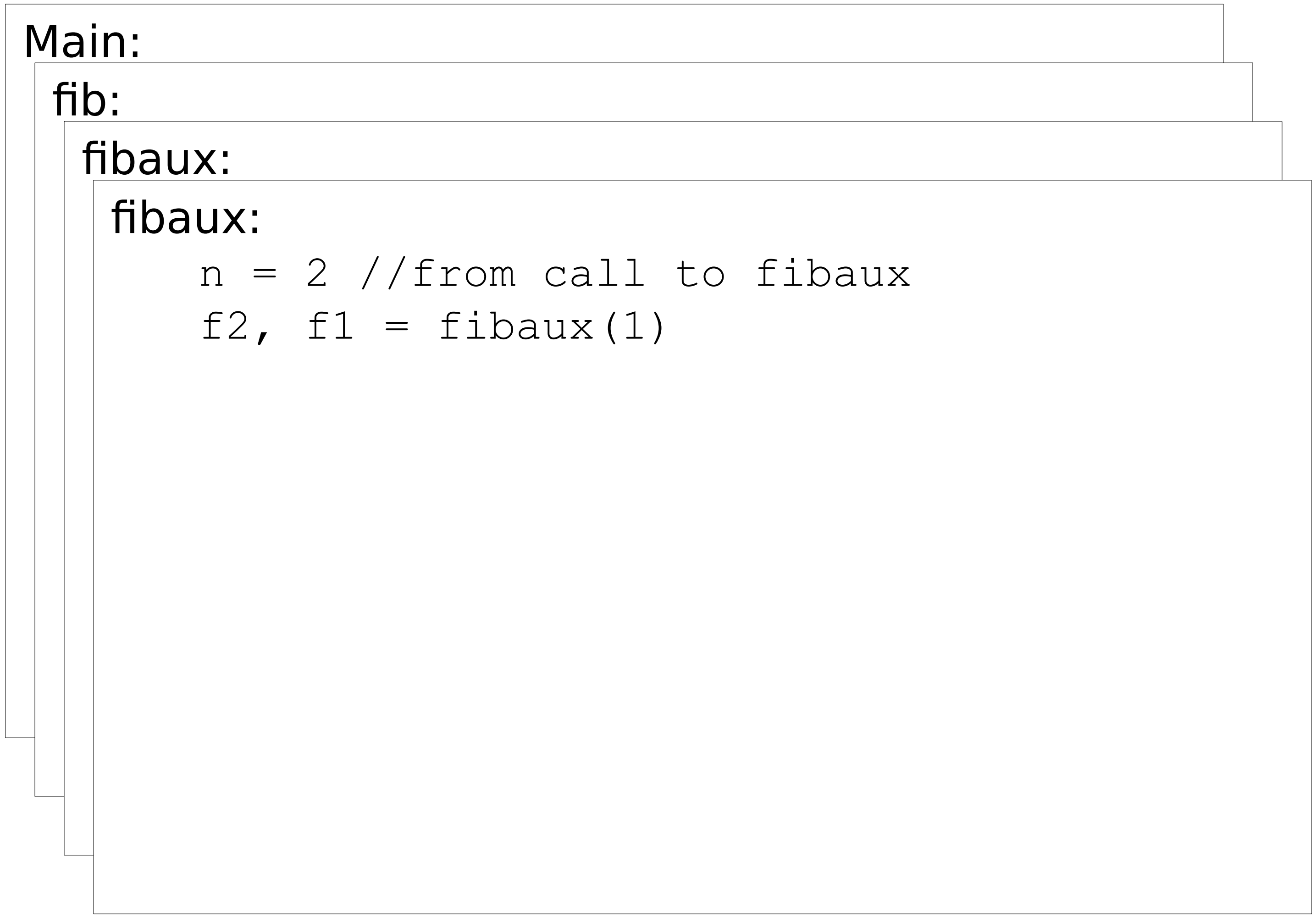
Main:

fib:

fibaux:

```
    n = 3 //from call to fibaux
    f2, f1 = fibaux(2)
```

Recursion: How it works



```
n = 2 //from call to fibaux
f2, f1 = fibaux(1)
```

Recursion: How it works

Main:

fib:

fibaux:

fibaux:

fibaux:

```
    n = 1 //from call to fibaux
    return 1, 0
```

Recursion: How it works

Main:

fib:

fibaux:

fibaux:

```
    n = 2 //from call to fibaux
    f2 = 1 // returned from fibaux(1)
    f1 = 0
    return f2 + f1, f2 // 1, 1
```

Recursion: How it works

Main:

fib:

fibaux:

```
    n = 3 //from call to fibaux
    f2 = 1 // returned from fibaux(2)
    f1 = 1
    return f2 + f1, f2 // 2, 1
```

Recursion: How it works

Main:

fib:

```
    n = 3 // from call to fib
    // fibaux(3) returns 2, 1
    return (2, 1)[0] // 2
```

Recursion: How it works

Main:

```
    fib(3) // returns 2
```

Possible problems of recursion: Inefficiency

- May have more overhead than similar iterative algorithms
  - Recursive calls involve repeatedly storing/removing data from the memory (the call stack)
- May perform excessive computation
  - If recomputing solutions for sub-problems
  - Example
    - Fibonacci numbers

Possible problems of recursion: Infinite loop

- Without proper definition of base case, the recursive calls will be trapped into infinite loop
- Every recursion must have at least one base case

Recursive Backtracking

Backtracking

- **backtracking**: A general algorithm for finding solution(s) to a computational problem by trying partial solutions and then abandoning them ("backtracking") if they are not suitable.
  - a "brute force" algorithmic technique (tries all paths; not clever)
  - often (but not always) implemented recursively

  Applications:
  - producing all permutations of a set of values
  - parsing languages
  - games: anagrams, crosswords, word jumbles, 8 queens
  - combinatorics and logic programming

Backtracking algorithms

A general pseudo-code algorithm for backtracking problems:

explore(**choices**):

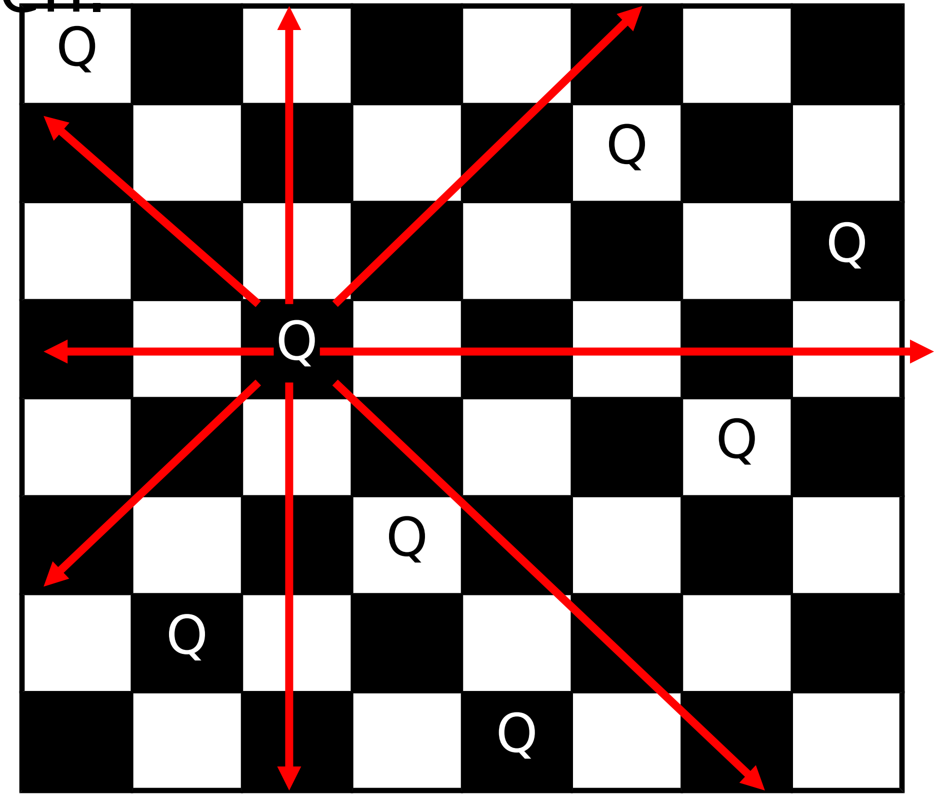
- while there are more **choices** to make:
  - Make a single choice **C** from the set of remaining choices.
  - explore the remaining **choices**.
    - This can be done recursively
  - Un-make choice **C**.
- having explored all choices
  - Stop.

Backtracking strategies

- When solving a backtracking problem, ask these questions:
  - What are the "choices" in this problem?
    - What is the "base case"? (How do I know when I'm out of choices?)
  - How do I "make" a choice?
    - Do I need to create additional variables to remember my choices?
    - Do I need to modify the values of existing variables?
  - How do I explore the rest of the choices?
    - Do I need to remove the made choice from the list of choices?
  - Once I'm done exploring the rest, what should I do?
  - How do I "un-make" a choice?

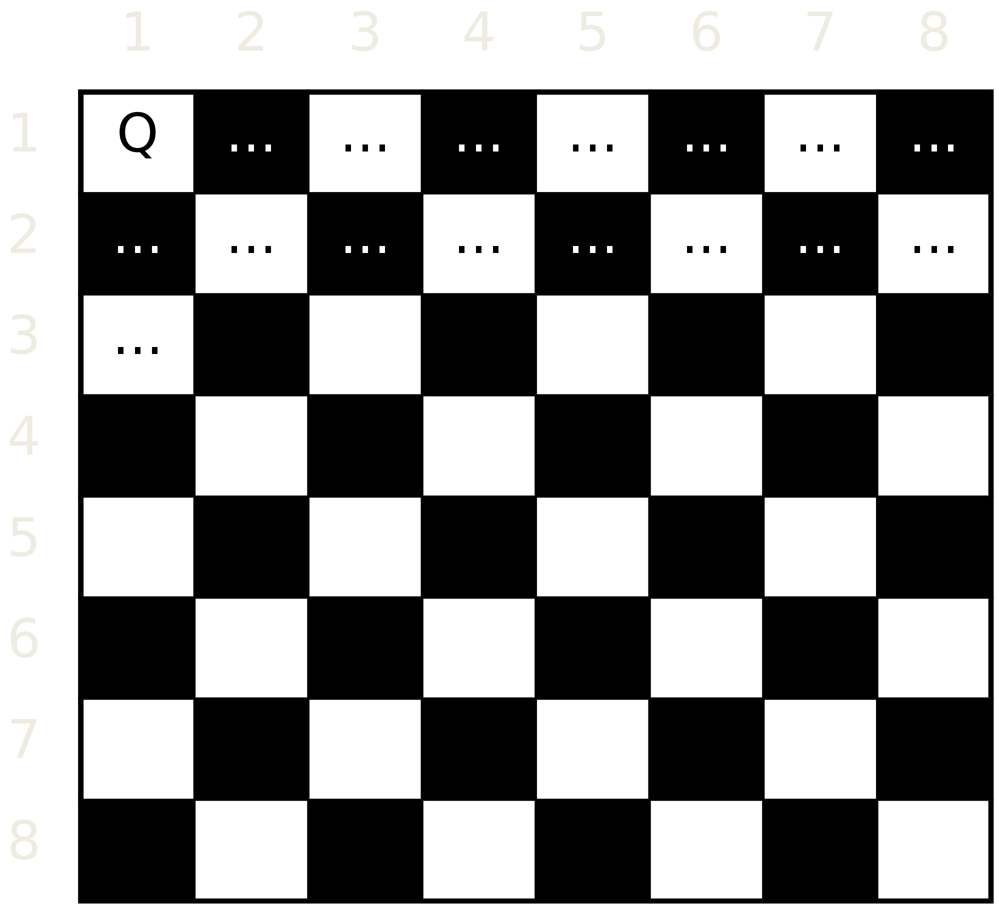
The "8 Queens" problem

- Consider the problem of trying to place 8 queens on a chess board such that no queen can attack another queen.
  - What are the "choices"?
  - How do we "make" or "un-make" a choice?
  - How do we know when to stop?



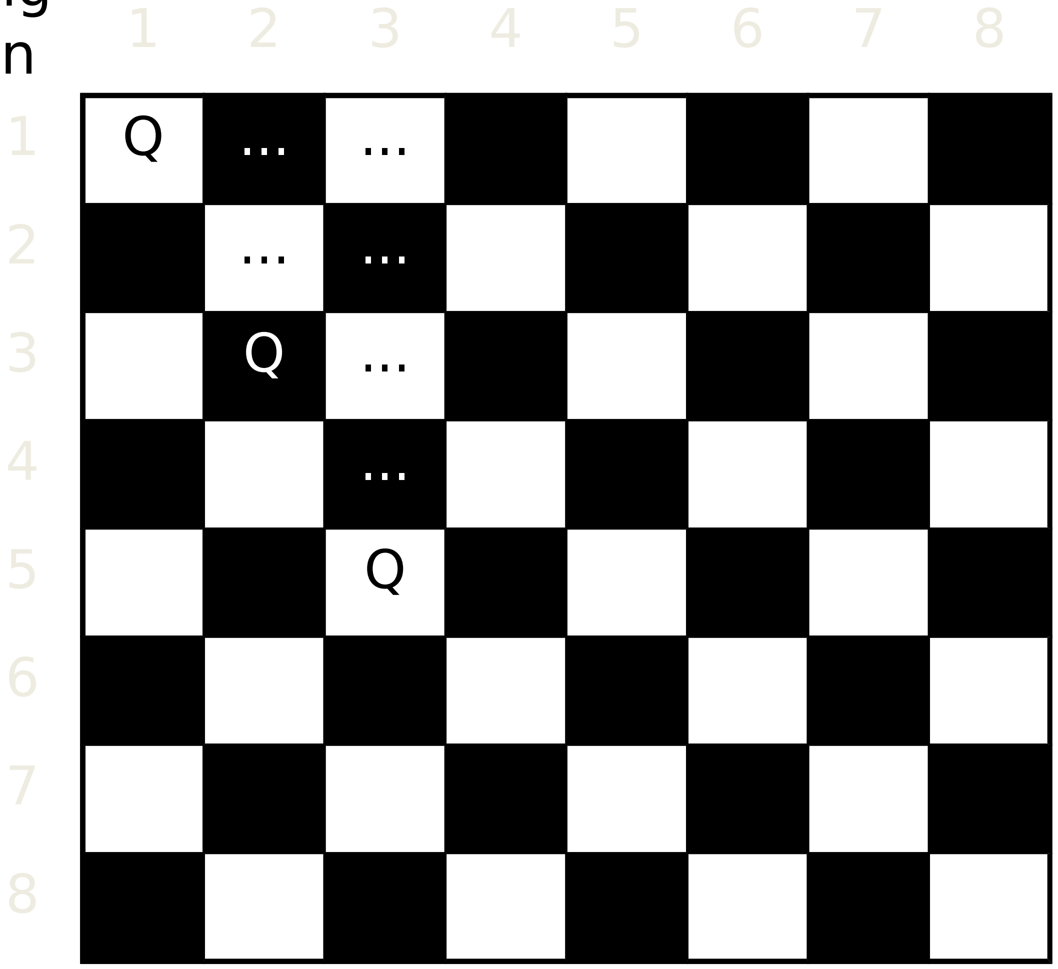
Naive algorithm

- for (each square on board):
  - Place a queen there.
  - Try to place the rest of the queens.
  - Un-place the queen.
  - How large is the solution space for this algorithm?
    - 64 * 63 * 62 * ...



Better algorithm idea

- Observation: In a working solution, exactly 1 queen must appear in each row and in each column.
  - Redefine a "choice" to be valid placement of a queen in a particular column.
  - How large is the solution space now?
    - 8 * 8 * 8 * ...



Exercise solution

```
// Search for a solution to the eight queens problem
// and report the first solution found.

public static void solveQueens() {
   // Board is initialized to all false
   boolean board[][] = new boolean[8][8];

   if (solve(board, 0)) {
      System.out.println("Solution:");
      printBoard(board);
   } else {
      System.out.println("No solution found.");
   }
}
```

Exercise solution, cont'd.

```
/* solve the board. Board must be an 8x8 array.
 * row must be in the range 0 <= row < 8 */
static boolean solve(boolean[][] board, int row) {
   if (row >= 8)
      return true;

   for (int col = 0; col < 8; col++) {
      if (isSafe(board, row, col)) {
         board[row][col] = true;
         if (solve(board, row + 1))
            return true;
         board[row][col] = false;
      }
   }
   return false;
}
```

isSafe checks to see if a queen at row, col is safe from attack by other queens.