RECURSION

COP 3502

What is recursion?

- First, let's talk about circular definitions.

> **mandiloquy.** (1) The conduct of maniloquy between nations; (2) Skill in doing this.

- Recursive definitions are just circular definitions
  - When we define something recursively we define it in terms of itself.
- But what makes a recursive definition of a problem X work, is that it shows how to define a big problem X into simpler versions of X.
  - Until at some point we reach a sub-problem small enough that we can solve it directly.

What is recursion?

- Definition: Any time the body of a function contains a call to the function itself.

- For example:
  - **$a^n = a * a^{n-1}$**
    - Defines an exponent into a smaller sub-problem,
    - until we get to the base case that we can solve on its own:
  - **$a^0 = 1$**

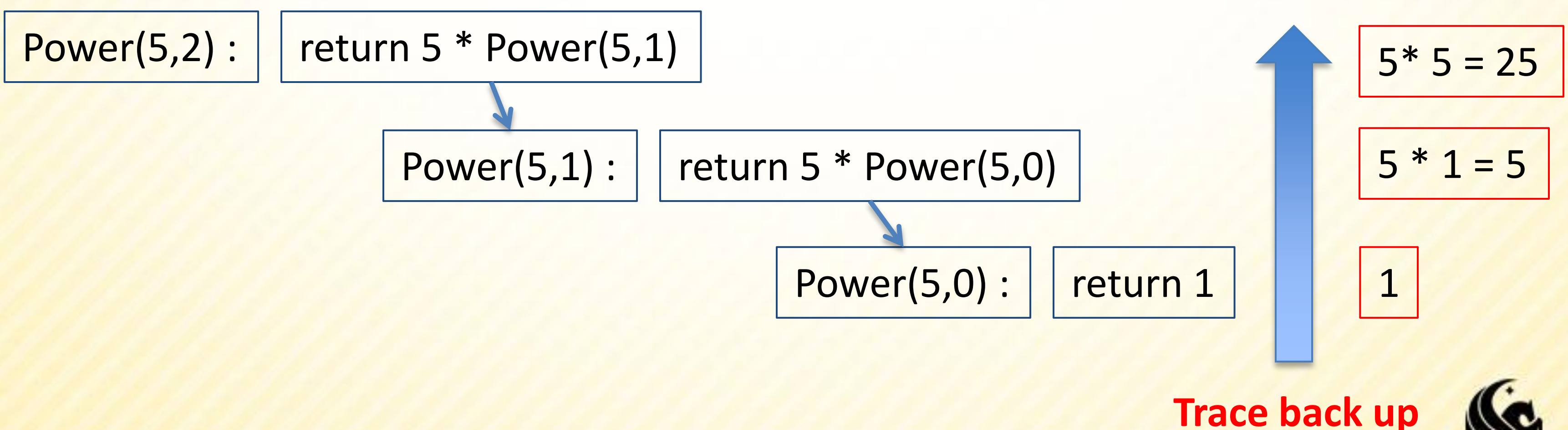
What is recursion?

- Since the definition of recursion is –
  - Any time the body of a function contains a call to the function itself.
  - How can we ever finish executing the original function?
- What this means is that some calls to the function **MUST NOT** result in a recursive call.
- Example:

```
// Pre-conditions:  exponent is >= to 0
// Post-conditions: returns base^exponent

int Power(int base, int exponent) {

      if (exponent == 0)
          return 1;
      else
          return (base*Power(base, exponent - 1);
}
```

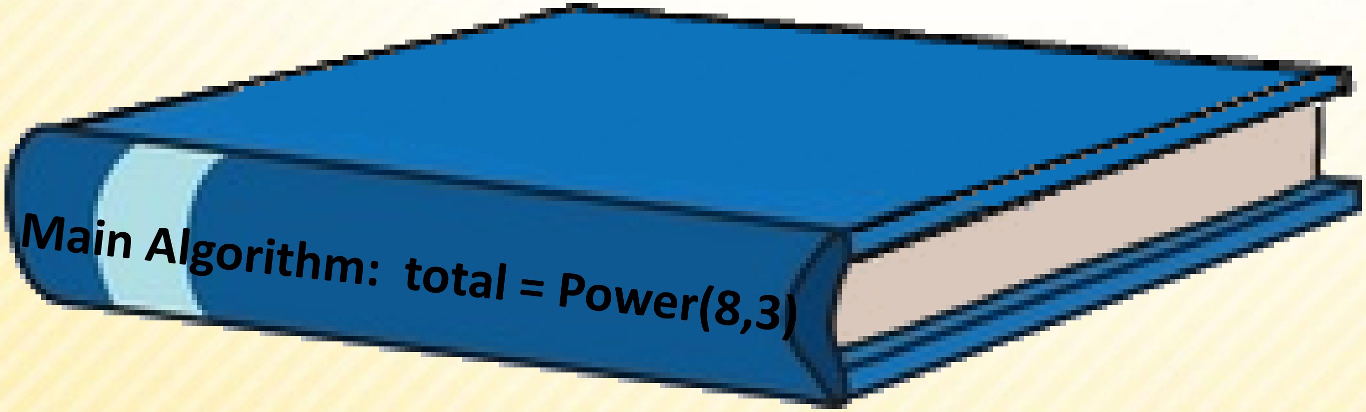
```
// Pre-conditions:  exponent is >= to 0
// Post-conditions: returns base^exponent

int Power(int base, int exponent) {

      if (exponent == 0)
          return 1;
      else
          return (base*Power(base, exponent – 1);
}
```

- To convince you that this works, let's look at an example:
  - Power(5,2):



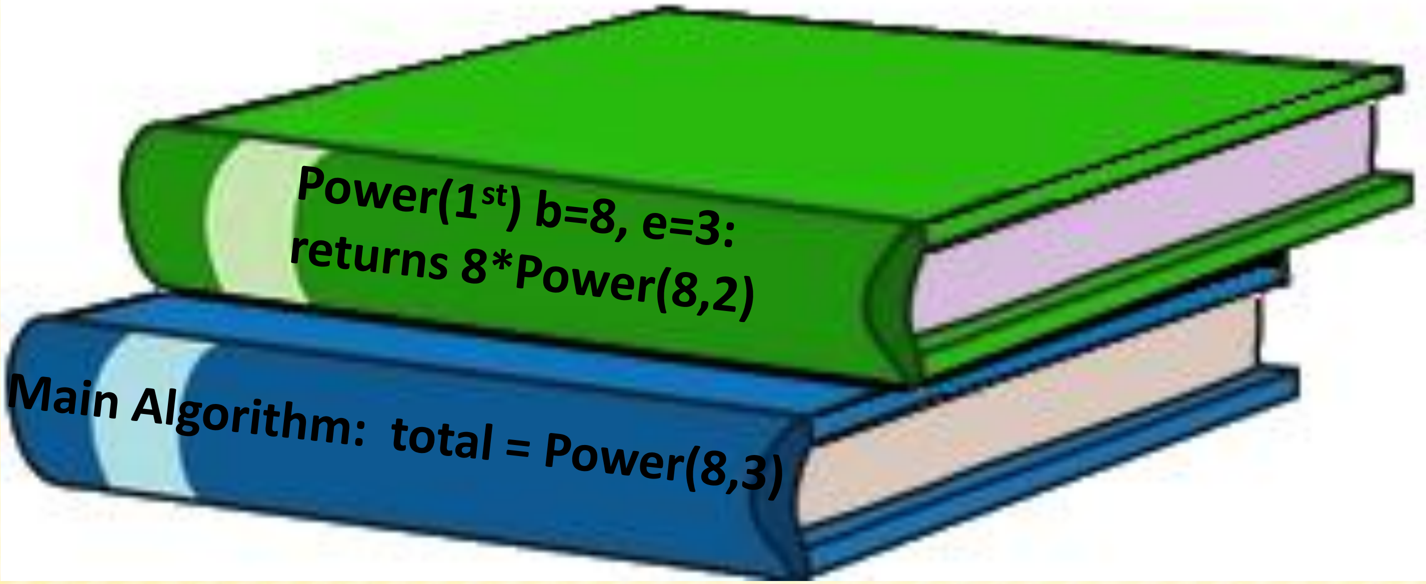
Using a Stack to Trace Recursive Code

- A stack is a construct that can be used to store and retrieve items
  - It works just like a stack of books:
    - The last book placed on top is the first one that must be removed.
    - OR a Last In, First Out (LIFO) system
  - Stacks can help us trace recursive functions.
  - Consider computing Power(8,3)
    - We can put a line of code from our main algorithm as the 1st item on the stack:



Using a Stack to Trace Recursive Code

- Now we need to compute the value of Power(8,3)…
  - So the function call Power(8,3) is placed above this statement in the stack:



Using a Stack to Trace Recursive Code

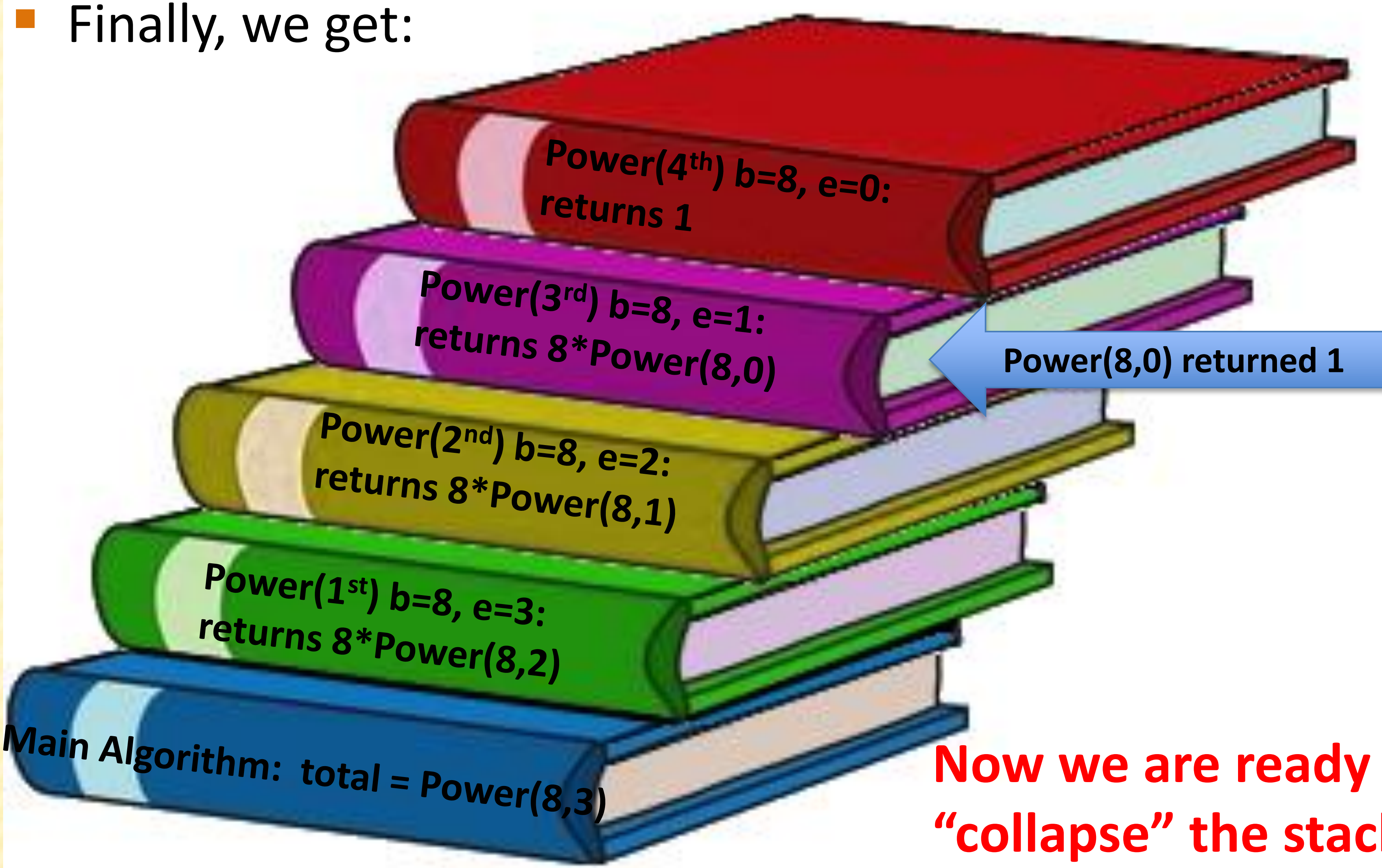
- Now we repeat the process…

Using a Stack to Trace Recursive Code

- Again…

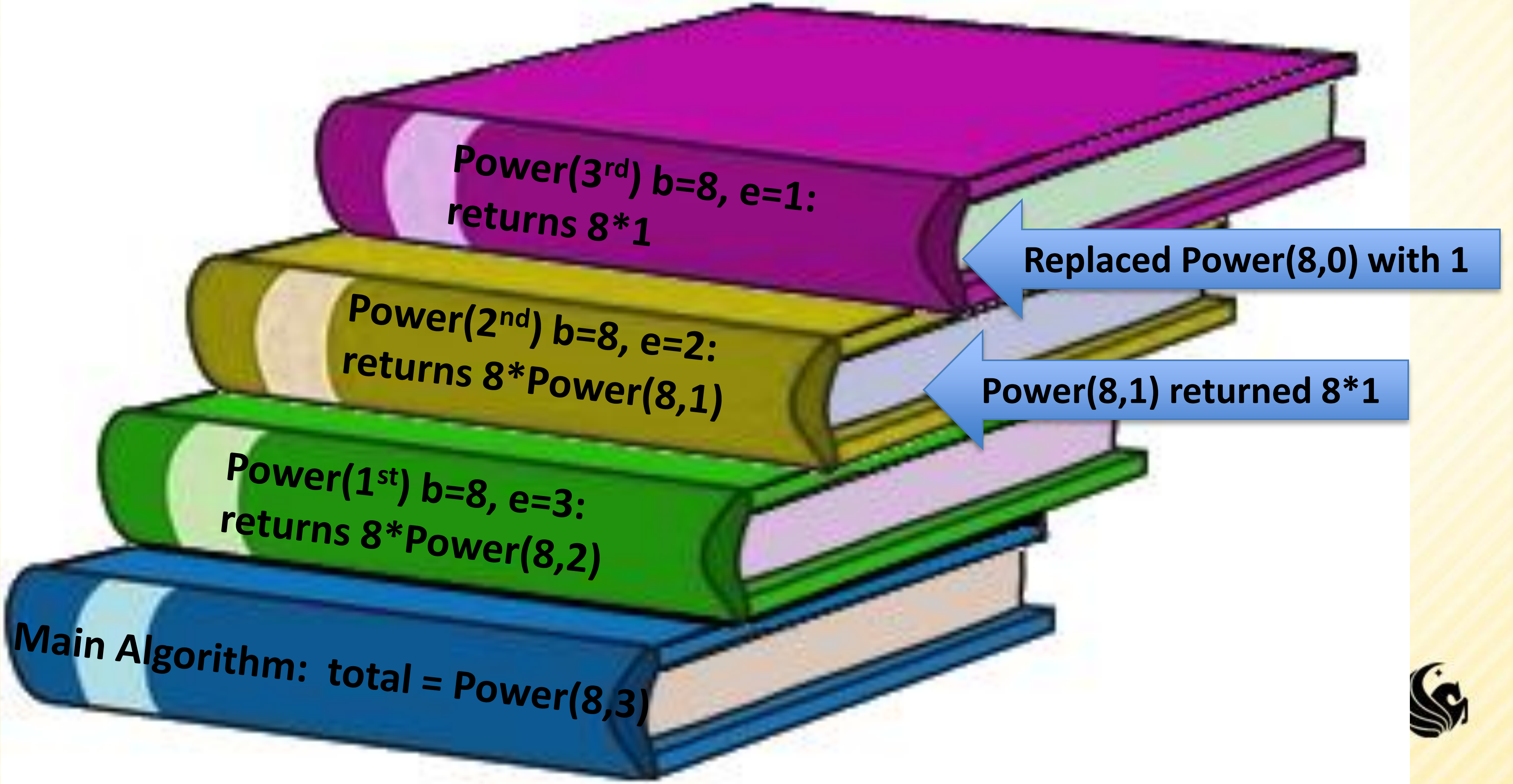
Using a Stack to Trace Recursive Code

- Finally, we get:

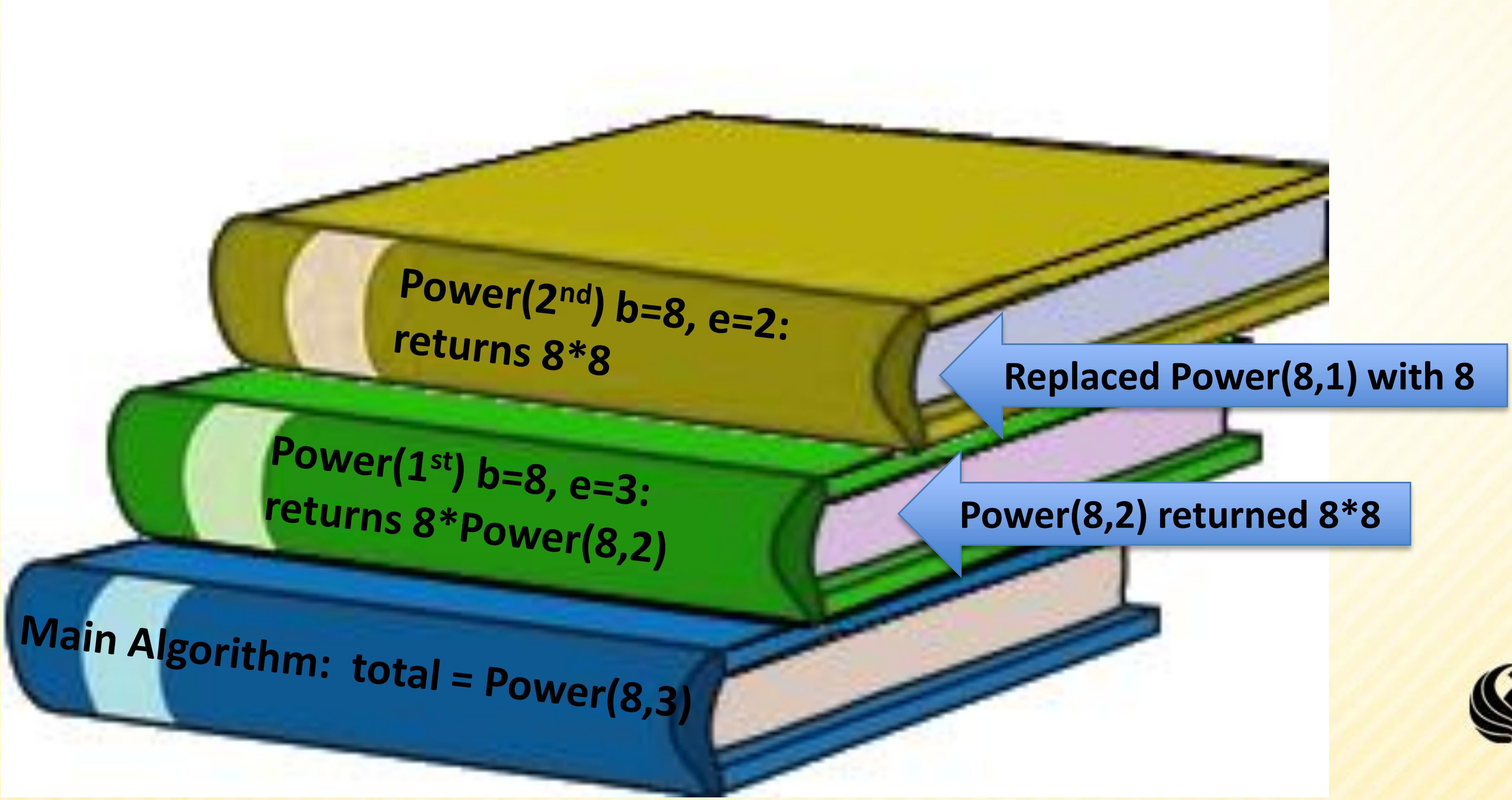


Now we are ready to "collapse" the stack!!

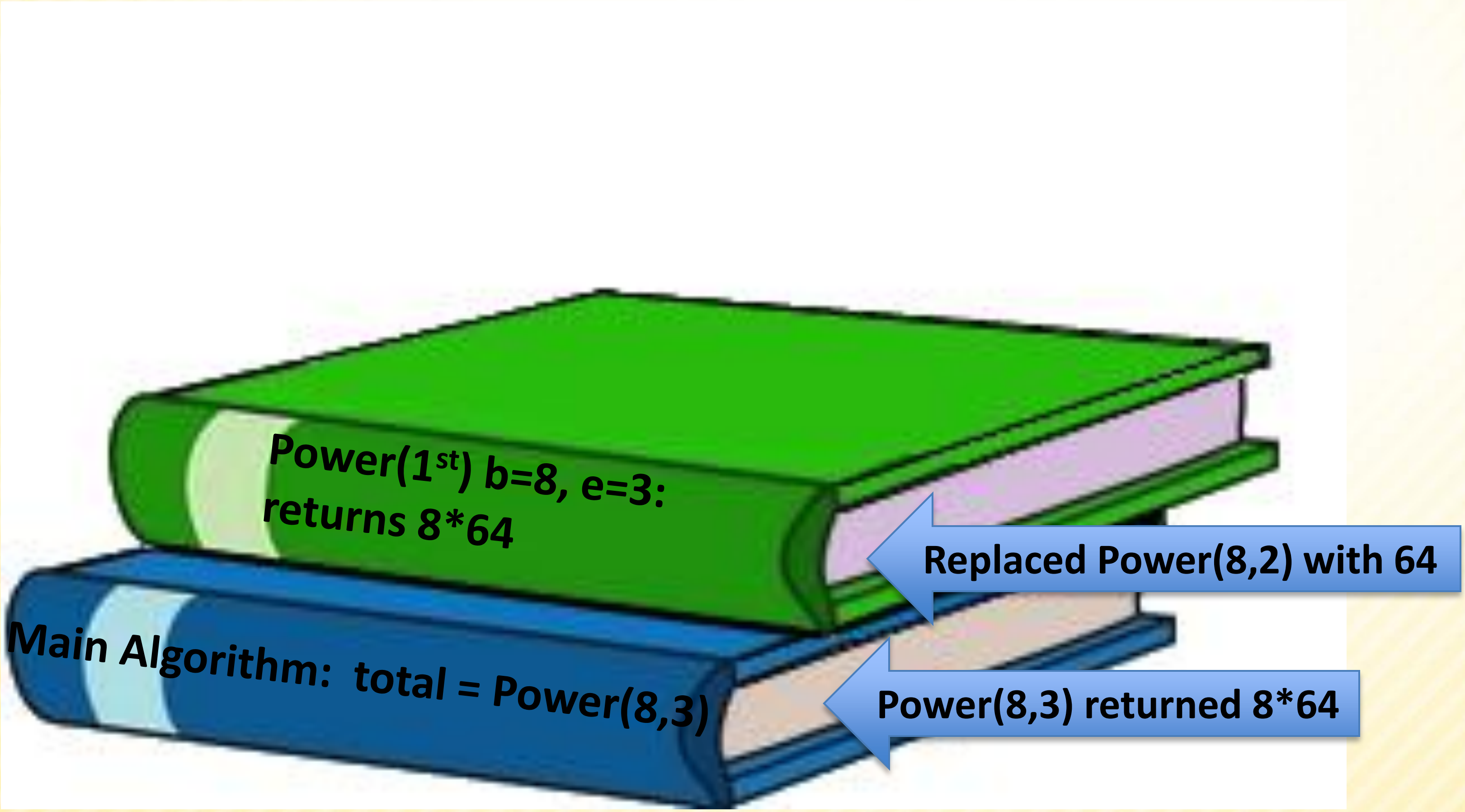
Using a Stack to Trace Recursive Code



Using a Stack to Trace Recursive Code



Using a Stack to Trace Recursive Code



Using a Stack to Trace Recursive Code

General Structure of Recursive Functions

- In general,
  - When we have a problem, we want to break it down into chunks, where one of the chunks is a smaller version of the same problem.
  - And eventually, we break down our original problem enough that, instead of making another recursive call, we can directly return the answer.
- So the general structure of a recursive function has a couple options:
  - Break down the problem further, into a smaller sub-problem
  - OR
  - the problem is small enough on its own, solve it

General Structure of Recursive Functions

- Here are 2 general constructs of recursive functions

```
if (termination condition) {
    DO FINAL ACTION
}
else {
    Take 1 step closer to
    terminating condition

    Call function RECURSIVELY
    on smaller sub-problem
}
```

Typically, functions that return values use this construct.

```
if (!termination condition) {
    Take 1 step closer to
    terminating condition

    Call function RECURSIVELY
    on smaller sub-problem
}
```

While void recursive function use the this construct.

Note: These are not the ONLY layouts of recursive programs, just common ones.

Example using construct 1

- Let's write a function that adds up all the squares of the numbers from m to n.
  - That is, given integers m and n, m <= n, we want to find:
  - $SumSquares(m,n) = m^2 + (m+1)^2 + \ldots + n^2$
  - For example: SumSquares(5,10) =
    - $5^2 + 6^2 + 7^2 + 8^2 + 9^2 + 10^2 = 355$
- Just so we're on the same page, let's write the iterative function:

```
int SumSquares(int m, int n)
{
    int i, sum;
    sum = 0;

    for (i = m; i <= n; i++)
        sum += i*i;

    return sum;
}
```

Example using construct 1

```
int SumSquares(int m, int n)
{
    if (m == n) {
        return m*m;
    }
    else {

        return m*m + SumSquares(m+1,n);

    }
}
```

Example Using Construct 2

- Let’s say we want to create a function that prints out a chart with the appropriate tips for meals ranging from first_val to lastval number of dollars, for every whole dollar amount.

```
#define TIP_RATE 0.15

void Tip_Chart(int first_val, int last_val)
{
    if (!(firstVal > lastVal)) {
        printf(“Ona meal of $%d”, first_val);
        printf(“you should tip $%f\n”, firstVal*TIP_RATE);

        Tip_Chart(first_val + 1, last_val);
    }
}
```

Recursion

- Why use recursion?
  - Some solutions are naturally recursive.
    - In these cases there might be less code for a recursive solution, and it might be easier to read and understand.
- Why NOT user recursion?
  - Every problem that can be solved with recursion can be solved iteratively.
  - Recursive calls take up memory and CPU time
    - Exponential Complexity – calling the Fib function uses 2n function calls.
  - Consider performance and software engineering principles.

Recursion Example

- Let's do another example problem – Fibonacci Sequence
  - 1, 1, 2, 3, 5, 8, 13, 21, …
- Let's create a function **`int Fib(int n)`**
  - we return the nth Fibonacci number
  - Fib(1) = 1, Fib(2) = 1, Fib(3) = 2, Fib(4) = 3, Fib(5) = 5, …
- What would our base (or stopping) cases be?

Fibonacci

- 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …

- Base (stopping) cases:
  - Fib(1) = 1
  - Fib(2) = 1,
- Then for the rest of the cases: Fib(n) = ?
  - Fib(n) = Fib(n-1) + Fib(n-2), for n>2

- So Fib(9) = ?
  - Fib(8) + Fib(7) = 21 + 13

Recursion - Fibonacci

- See if we can program the Fibonacci example…