Recursion

Outline

- Introduction (What is Recursion?)
- How to trace a recursive function call?
- Examples
  - Fibonacci Numbers
  - Finding the largest number in an array (Optional)
  - Tower of Hanoi (Optional)

Introduction

- ***Recursion*** – a method of defining functions in which the function being defined may be used within its own definition.

- e.g.:

$$\left.\begin{array}{ll} f(N) = N * f(N\text{-}1) & , \text{ if } N \geq 1 \\ f(N) = 1 & , \text{ if } N = 0 \end{array}\right\} \text{Recursive solution (Not in C syntax)}$$

  - What is the value of f(3)?
  - What is this function?

Implementing `f()` as a Recursive Function

```c
int f(int n) {
    if (n == 0)
        return 1;
    return n * f(n-1);
}

int main(void) {
    printf("f(3) = %d\n", f(3));
    return 0;
}
```

What happen when a function calls "itself"?

f(3)

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

```
n = 3
```

Initially, `main()` calls `f(3)`.

In this function call, **n** receives **3** from `main()`.

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

n = 3

f(2)

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

n = 2

Conceptually, the function is not actually calling "itself" but a "clone" of itself.

Each "clone" of the function has its own local variables.

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

```
n = 3
```

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

```
n = 2
```

f(1)

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

```
n = 1
```

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

```
n = 3
```

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

```
n = 2
```

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

```
n = 1
```

f(0)

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

```
n = 0
```

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

```
n = 3
```

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

```
n = 2
```

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

```
n = 1
```

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

```
n = 0
```

returns **1**

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

n = 3

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

n = 2

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

n = 1

returns

1 * f(0)

= 1 * 1

= 1

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

n = 3

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

2 1

n = 2

returns

2 * f(1)

= 2 * 1

= 2

```
int f(int n) {
  if (n == 0) return 1;
  return n * f(n-1);
}          3     2
```

```
n = 3
```

returns

`3 * f(2)`

`= 3 * 2`

`= 6`

Finally, 6 is returned to `main()`.

Recursion – Stack Overflow

- One of the important components of a recursive function is the **termination condition**.

```
int f(int n) {
  if (n == 0) return 1;  // Termination condition
  return n * f(n-1);
}
```

- When a recursion cannot terminate, the computer will eventually run out of "stack space" in the memory and generate a runtime error called **stack overflow**.
  - You can experience the error by removing the if statement in the above code.

Exercise: What's the output?

```c
void print(int x) {
    printf("Before: %d\n", x);  // Before calling print()

    if (x <= 0)
        return;
    print( x - 1 );

    printf("After: %d\n", x);   // After calling print()
}

int main() {
    print(2);
    return 0;
}
```

```
Before: 2
Before: 1
Before: 0
After: 1
After: 2
```

Example: Fibonacci Number

- The Fibonacci number introduction video:
  - http://www.ted.com/talks/arthur_benjamin_the_magic_of_fibonacci_numbers
- Fibonacci number is a sequence of number:
  - The *n*-th number in the sequence **depends on the previous two numbers** in the same sequence.
  - e.g., 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …

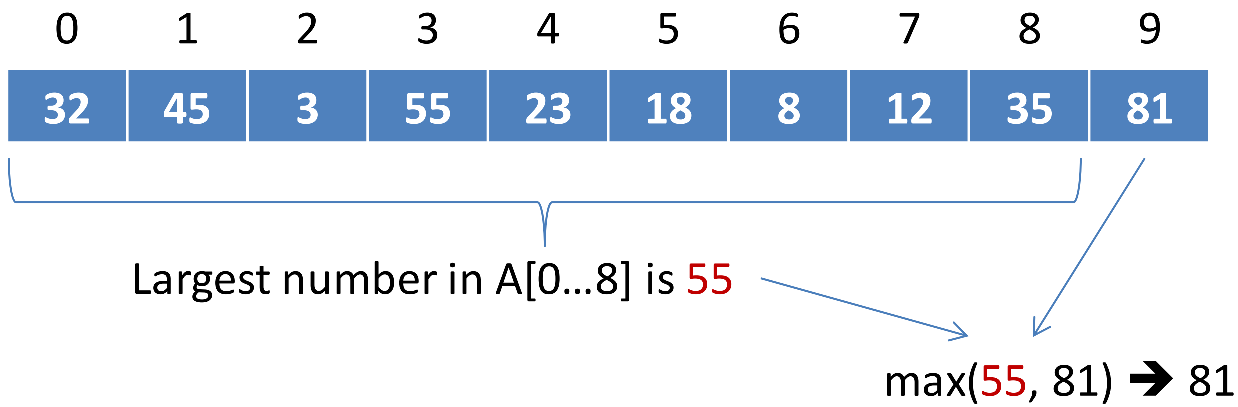
```
n-th Fibonacci number: f(n)

if n is 0:  return 0;
if n is 1:  return 1;
if n is > 1:
    return f(n-1) + f(n-2);
```

We leave the implementation to you as an exercise.

Example: Finding the largest number in an array of N integers **(Optional)**

- Let A[0…M] = the sub-array containing elements A[0], A[1], …, A[M]

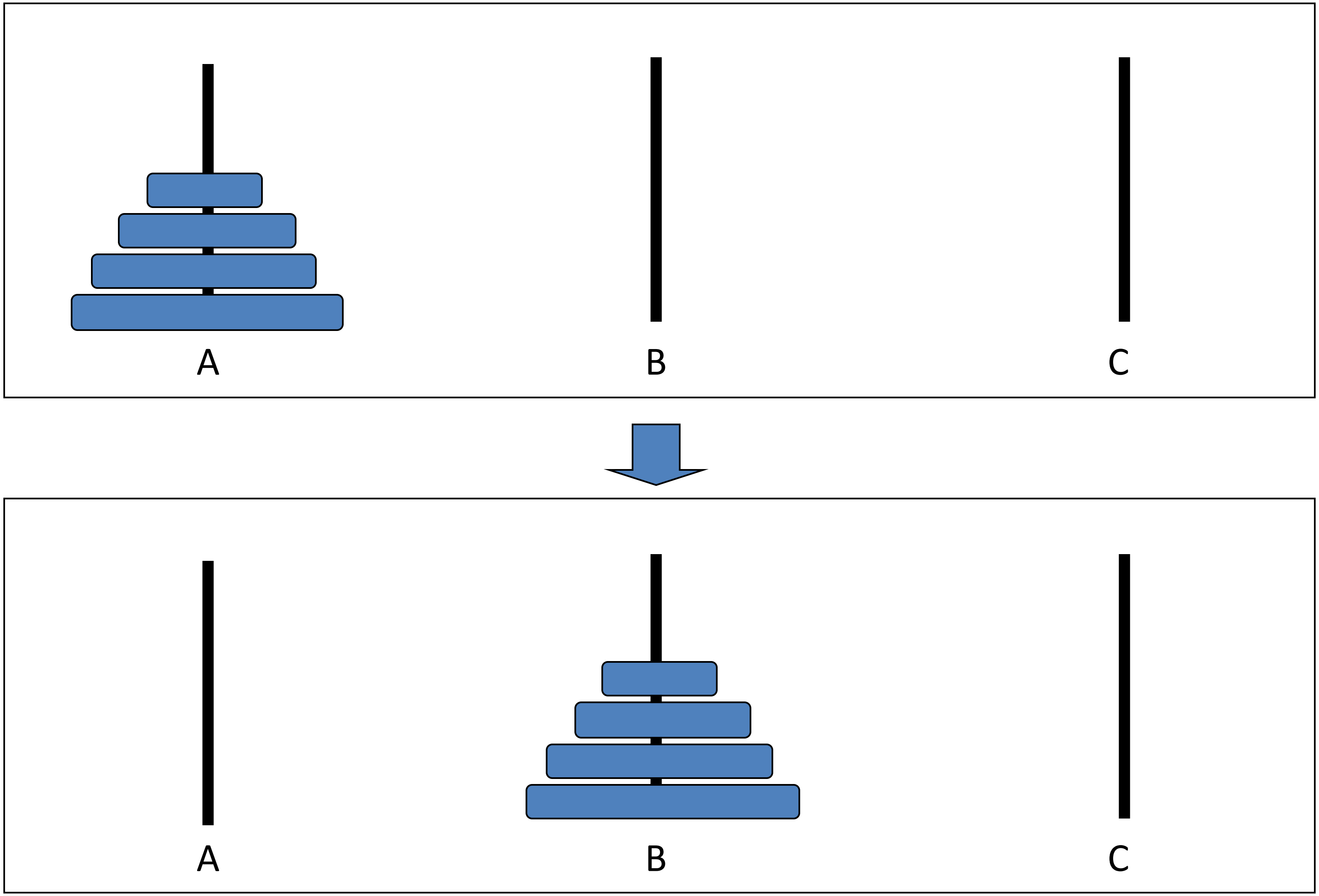
- We can define a recursive function to find the largest element in an array A (with N elements) as
  - largest(A[0…N-1]) = max(largest(A[0…N-2]), A[N-1])
  - largest(A[0…0]) = A[0]

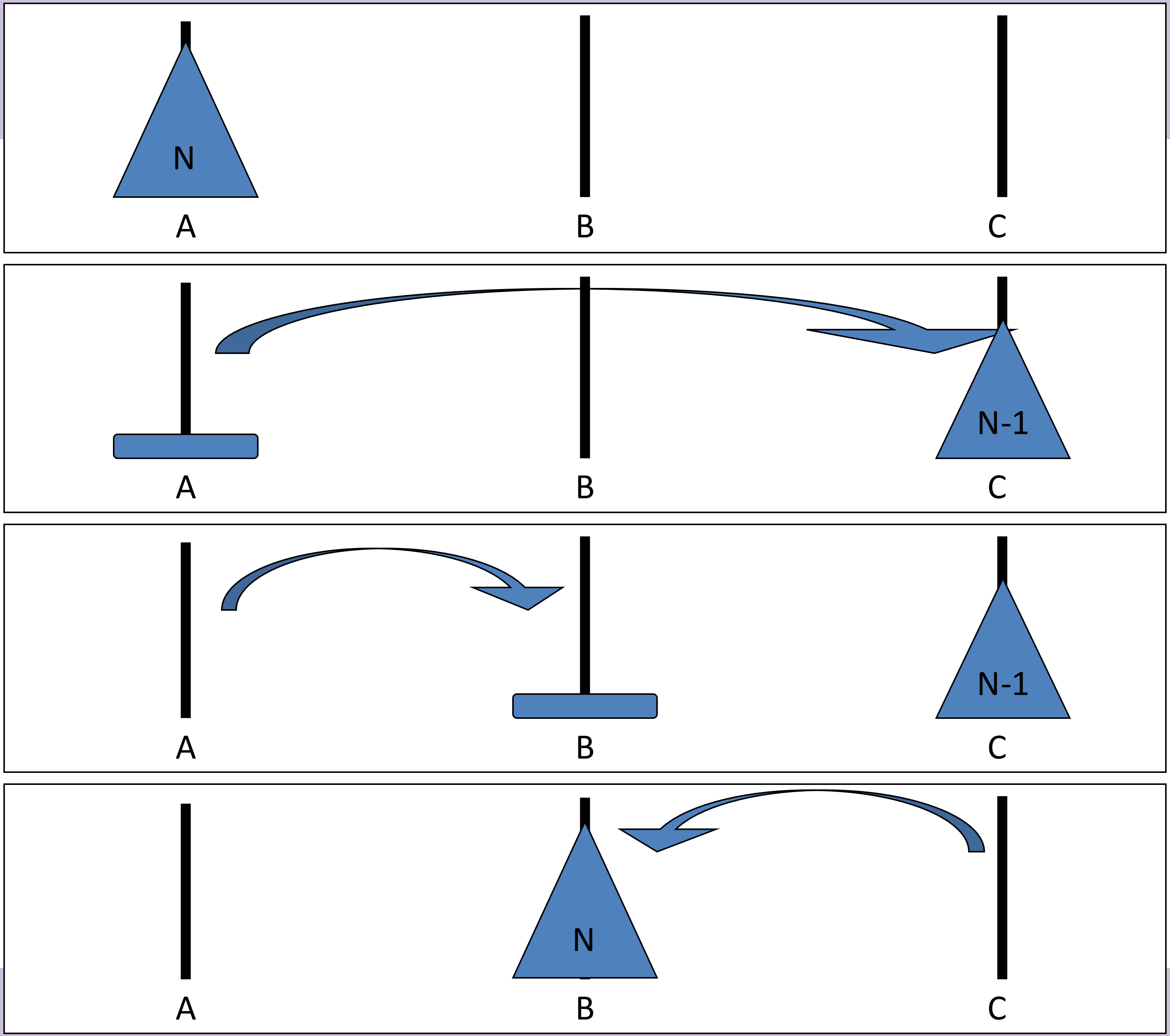


Example: Finding the largest number in an array of N integers **(Optional)**

```
// N is the # of elements to be processed
int largest(int a[], int N) {
    int tmp;

    if (N == 1)
         return a[0];

    // Find the largest in A[0...N-2] (first N-1 elements)
    tmp = largest(a, N-1);

    // Compare the last element to the largest element in A[0...N-2]
    if (a[N-1] > tmp)
        return a[N-1];
    else
        return tmp;
}
```

Example: Tower of Hanoi **(Optional)**




N
A
B
C
N-1
A
B
C
N-1
A
B
C
N
A
B
C

Example: Tower of Hanoi **(Optional)**

```c
// Output all the moves needed to move N discs from "src" to
// "dst" using "tmp" as temporary stick.
void move(char src, char dst, char tmp, int N) {
    if (N == 1) {
        printf("Move from %c to %c\n", src, dst);
        return;
    }

    move(src, tmp, dst, N - 1);
    printf("Move from %c to %c\n", src, dst);
    move(tmp, dst, src, N-1);
}

int main(void) {
   move('A', 'B', 'C', 5);
   return 0;
}
```

This example is used to illustrate the elegance of a recursive solution.

Recursion – Pros and Cons

- Some solutions are easier or clearer to express as recursive solutions.
  - e.g., Tower of Hanoi

- Shortcomings: Slower in performance (compare to iterations. i.e. loops)
  - e.g.: Fibonacci numbers: f(n) = f(n-1) + f(n-2)
  - Each level of recursion doubles the number of function calls
    - 30^{th} number = 2^{30} ~ 4 billion function calls

Summary

- Understand how to trace recursive function calls
- Know how to implement a recursive function in C when presented a "simple" recursive solution for a problem

Reading Assignment

- C: How to Program, 8th ed, Deitel and Deitel
- Chapter 5 C Functions
  - Section 5.7: Function Call Stack and Stack Frames
  - Sections 5.14 – 5.16: Recursion and Example