Writing the Fibonacci Sequence

First, let’s go back to Fibonacci.

We had:  
\[ F(0) = 1 \]
\[ F(1) = 1 \]
\[ F(N) = F(N – 2) + F(N – 1) \]

Let’s write Fibonacci as a C function:

```c
int32_t fib (int32_t N);
```

Fibonacci as a Recursive Function

```c
int32_t fib (int32_t N)
{
    if (0 == N || 1 == N) {
        return 1;
    }
    return (fib (N – 1) +
            fib (N – 2));
}
```

Pitfall: Recursion May Not Be the Answer

Being able to write a function recursively does not imply that doing so is a good idea.

Consider calling `fib (5)`, for example.

How many times is `fib` called as a result?
Recursive Fibonacci Does Too Much Work

There is a closed form solution... And iteration is not hard.

A Brief Iterative Implementation of Fibonacci

```c
int32_t fib (int32_t N)
{
    int32_t i, j, k, t;
    for (i = j = k = 1; N > k;
        t = j, j = j + i, i = t,
        k++) {}
    return j;
}
```

Linear Recursive Fibonacci Using a `static` Variable

```c
int32_t fib (int32_t N)
{
    static int32_t fibval;
    int32_t pred, result;
    if (2 > N) {
        fibval = 1;
        return 1;
    }
    pred = fib (N - 1);
    result = pred + fibval;
    fibval = pred;
    return result;
}
```

Binary Search Can Also Be Written Recursively

```c
int32_t binary_search
(int32_t array[],
 int32_t low,
 int32_t high,
 int32_t value)
{
    the array to search
    the part of the array
    in which to look
    the value to find
    return result;
}
```
Recursive Version is Slightly Simpler

(The code is slightly simpler.)

```c
int32_t mid;
if (high < low) { return -1; }
mid = low + (high - low) / 2;
```

Recurse with Modified Bounds When Not Found

```c
if (value == array[mid]) {
    return mid;  // Found!
}
if (value < array[mid]) {
    return binary_search
          (array, low, mid - 1, value);
}
return binary_search
       (array, mid + 1, high, value);
```

Some Types of Recursion Can Be Compiled Away

When recursion
- happens only at the end of a function,
- in other words: return <recursive call>,
- it is called tail recursion.
Binary search is an example of tail recursion.
A good optimizing compiler
- can transform tail recursion
- into an iterative version,
- avoiding use of extra stack frames.

Let’s Do an Example Together

Help me solve this problem recursively...

Task:
- print a string backwards and
- return its length (not counting NUL).
Let’s call the function `print_reverse`.

What arguments should be passed?
a (constant) string

What should the return type be? `int32_t`
What Comes First in a Recursive Function?

What comes first?

stopping conditions

When do we stop?

at end of string (NUL)

How long is s in that case?

0

What About This “Node?” What About Children?

If the string isn’t empty, we need to

◦ print one character, and

◦ call print_reverse with the rest of the string.

In what order?

Call first, then print.

We also need a local variable to store the return value from the call.

How Do We Make the Recursive Call?

What character should be printed?

What should be passed to the recursive call?

rest = print_reverse (s + 1);

int32_t print_reverse
(const char* s)
{
    int32_t rest;
    if (‘\0’ == *s) {
        return 0;
    }
    rest = print_reverse (s + 1);
}

What Do We Print?

int32_t print_reverse
(const char* s)
{
    int32_t rest;
    if (‘\0’ == *s) {
        return 0;
    }
    rest = print_reverse (s + 1);
    printf (“%c”, *s);
**What Do We Return?**

```c
int32_t print_reverse(const char* s)
{
    int32_t rest;
    if ('\0' == *s) {
        return 0;
    }
    rest = print_reverse(s + 1);
    printf("%c", *s);
    return (rest + 1);
}
```

**Reference Version of print_reverse**

```c
int32_t print_reverse(const char* s)
{
    int32_t rest;
    if ('\0' == *s) {
        return 0;
    }
    rest = print_reverse(s + 1);
    printf("%c", *s);
    return (rest + 1);
}
```

**Let's See How a Recursive Function Works in Detail**

Let's execute a call:

```
print_reverse("Now")
```

With ... stack frames!

This visualization will probably be the last time that we see stack frames in class.

Feel free to get nostalgic.

**Stack Frame for print_reverse (Call Depth 1)**

```
call depth 1
p_r ("Now")
```

Abbreviated as p_r here.

Say that the N is at x4000.

What are the values when the function starts?

```
R5,R6= rest (bits)
linkage | ret. val (bits)
     s (x4000)
```
int32_t print_reverse(const char* s) {
    int32_t rest;
    if ('\0' == *s) {  // Check for NUL.
        return 0;
    }
    rest = print_reverse(s + 1);
    printf("%c", *s);
    return (rest + 1);
}

What is Passed to the Recursive Call?

int32_t print_reverse(const char* s) {
    int32_t rest;
    if ('\0' == *s) {  // 'N' is not NUL, so don't return yet.
        return 0;
    }
    rest = print_reverse(s + 1);
    printf("%c", *s);
    return (rest + 1);
}

Call print_reverse again with (s + 1).
Calling `print_reverse` (Call Depth 2)

```c
int32_t print_reverse(const char* s)
{
    int32_t rest;
    if ('\0' == *s) { Check for NUL.
        return 0;
    }
    rest = print_reverse (s + 1);
    printf ("%c", *s);
    return (rest + 1);
}
```

What Happens First?

```
int32_t print_reverse(const char* s)
{
    int32_t rest;
    if ('\0' == *s) { Check for NUL.
        return 0;
    }
    rest = print_reverse (s + 1);
    printf ("%c", *s);
    return (rest + 1);
}
```
What Happens Next?

```c
int32_t print_reverse(const char* s) {
    int32_t rest;
    if ('\0' == *s) {
        return 0;
    }
    rest = print_reverse(s + 1);
    printf("%c", *s);
    return (rest + 1);
}
```

Call `print_reverse` again with `(s + 1)`.

---

Calling `print_reverse` (Call Depth 3)

- Call depth 1:
  ```c
  p_r ("Now")
  ```
- Call depth 2:
  ```c
  p_r ("ow")
  ```
- Call depth 3:
  ```c
  p_r ("w")
  ```

The w is at x4002.

Stack Frame for `print_reverse` (Call Depth 3)

- Call depth 1:
  ```c
  p_r ("Now")
  ```
- Call depth 2:
  ```c
  p_r ("ow")
  ```
- Call depth 3:
  ```c
  p_r ("w")
  ```

What Happens First?

```c
int32_t print_reverse(const char* s) {
    int32_t rest;
    if ('\0' == *s) {
        return 0;
    }
    rest = print_reverse(s + 1);
    printf("%c", *s);
    return (rest + 1);
}
```

- Check for NUL.
- ... same as before, of course!
- Then call `print_reverse` again.
Calling `print_reverse` (Call Depth 4)

The argument is the empty string at x4003.

Stack Frame for `print_reverse` (Call Depth 4)

What Happens First Now?

```c
int32_t print_reverse(const char* s) {
    int32_t rest;
    if ('\0' == *s) { Found NUL, so return 0.
        return 0;
    }
    rest = print_reverse (s + 1);
    printf ("%c", *s);
    return (rest + 1);
}
```

Write 0 into Return Value Slot
What Happens When `print_reverse` Returns?

```c
int32_t print_reverse(const char* s) {
    int32_t rest;
    if ('\0' == *s) {
        return 0;
    }
    rest = print_reverse (s + 1);
    printf ("%c", *s);
    return (rest + 1);
}
```

On return, return value is written into `rest`.

© 2018 Steven S. Lumetta. All rights reserved.

slide 37

ECE 220: Computer Systems & Programming

Tear Down Stack Frame

```
call depth 1
P_R ("Now")
```

```
call depth 2
P_R ("ow")
```

```
call depth 3
P_R ("w")
```

```
call depth 4
P_R ("n")
```

```
R5,R6→ rest (bits)
```

```
...and write 0 into rest.
```

```
s (x4003)
```

```
R5,R6→ rest (0)
```

```
linkage | ret. val (bits)
```

```
s (x4002)
```

```
R5,R6→ rest (bits)
```

```
linkage | ret. val (bits)
```

```
s (x4001)
```

```
R5,R6→ rest (bits)
```

```
linkage | ret. val (bits)
```

```
s (x4000)
```

```
rest (bits)
```

```
linkage | ret. val (bits)
```

```
s (x4002)
```

```
rest (bits)
```

```
linkage | ret. val (bits)
```

```
s (x4001)
```

```
rest (bits)
```

```
linkage | ret. val (bits)
```

```
s (x4000)
```

```
rest (bits)
```

```
linkage | ret. val (bits)
```

```
s (x4000)
```

```
returns 0
```

```
s (x4000)
```

```
returns 0
```

```
s (x4003)
```

```
returns 0
```

```
s (x4002)
```

```
s (x4001)
```

© 2018 Steven S. Lumetta. All rights reserved.

slide 38

ECE 220: Computer Systems & Programming

What Happens Next?

```c
int32_t print_reverse(const char* s) {
    int32_t rest;
    if ('\0' == *s) {
        return 0;
    }
    rest = print_reverse (s + 1);
    printf ("%c", *s);
    return (rest + 1);
}
```

Print `s`.

© 2018 Steven S. Lumetta. All rights reserved.

slide 39

ECE 220: Computer Systems & Programming

Print Character at `s`

```
call depth 1
P_R ("Now")
```

```
What is `s`? x4002
```

```
What is stored at x4002? 'w'
```

```
call depth 2
P_R ("ow")
```

```
R5,R6→ rest (0)
```

```
linkage | ret. val (bits)
```

```
s (x4002)
```

```
R5,R6→ rest (bits)
```

```
linkage | ret. val (bits)
```

```
s (x4001)
```

```
R5,R6→ rest (bits)
```

```
linkage | ret. val (bits)
```

```
s (x4000)
```

```
returns 0
```

```
s (x4000)
```

```
returns 0
```

```
s (x4000)
```

```
returns 0
```

```
s (x4001)
```

```
returns 0
```

```
s (x4000)
```

© 2018 Steven S. Lumetta. All rights reserved.

slide 40

ECE 220: Computer Systems & Programming
int32_t print_reverse(const char* s)
{
    int32_t rest;
    if ('\0' == *s) {
        return 0;
    }
    rest = print_reverse (s + 1);
    printf ("%c", *s);
    return (rest + 1);
}

What Happens Next?

Return (rest + 1)

Write 1 into Return Value Slot

What Happens When print_reverse Returns?
**Tear Down Stack Frame**

```latex
\begin{align*}
\text{call depth 1} & \quad \text{p_r} \quad \text{"Now"} \\
\text{call depth 2} & \quad \text{p_r} \quad \text{"ow"} \\
\text{call depth 3} & \quad \text{p_r} \quad \text{"w"} \\
\text{call depth 4} & \quad \text{p_r} \quad \text{"n"}
\end{align*}
```

...and write 1 into rest.

```latex
\text{R5,R6} \\
\text{rest (bits)} \\
\text{linkage} \quad \text{ret. val (bits)} \\
\text{s (x4002)} \\
\text{rest (0)} \\
\text{linkage} \quad \text{ret. val (bits)} \\
\text{s (x4001)} \\
\text{rest (bits)} \\
\text{linkage} \quad \text{ret. val (bits)} \\
\text{s (x4000)}
```

**What Happens Next?**

```c
int32_t print_reverse(const char* s)
{
    int32_t rest;
    if ('\0' == *s)
    {
        return 0;
    }
    rest = print_reverse (s + 1);
    printf ("%c", *s);
    return (rest + 1);  // Print 's.
}
```

---

**Print Character at s**

```latex
\begin{align*}
\text{call depth 1} & \quad \text{p_r} \quad \text{"Now"} \\
\text{call depth 2} & \quad \text{p_r} \quad \text{"ow"} \\
\text{call depth 3} & \quad \text{p_r} \quad \text{"w"} \\
\text{call depth 4} & \quad \text{p_r} \quad \text{"n"}
\end{align*}
```

What is s? x4001

What is stored at x4001? 'o'

```latex
\text{R5,R6} \\
\text{rest (1)} \\
\text{linkage} \quad \text{ret. val (bits)} \\
\text{s (x4001)} \\
\text{rest (bits)} \\
\text{linkage} \quad \text{ret. val (bits)} \\
\text{s (x4000)}
```

**What Happens Next?**

```c
int32_t print_reverse(const char* s)
{
    int32_t rest;
    if ('\0' == *s)
    {
        return 0;
    }
    rest = print_reverse (s + 1);
    printf ("%c", *s);
    return (rest + 1);  // Output is now "wo".
}
```

---

```c
\text{R5,R6} \\
\text{rest (1)} \\
\text{linkage} \quad \text{ret. val (bits)} \\
\text{s (x4001)} \\
\text{rest (bits)} \\
\text{linkage} \quad \text{ret. val (bits)} \\
\text{s (x4000)}
```

Return (rest + 1).
What is `rest`?

Adding 1, we obtain 2.

Returns

```c
int32_t print_reverse(const char* s) {
    int32_t rest;
    if ('\0' == *s) { return 0; }
    rest = print_reverse (s + 1);
    printf ("%c", *s);
    return (rest + 1);
}
```
int32_t print_reverse(const char* s) {
    int32_t rest;
    if ('\0' == *s) {
        return 0;
    }
    rest = print_reverse(s + 1);
    printf("\"c\", *s);
    return (rest + 1);
}

What Happens Next?

int32_t print_reverse(const char* s) {
    int32_t rest;
    if ('\0' == *s) {
        return 0;
    }
    rest = print_reverse(s + 1);
    printf("\"c\", *s);
    return (rest + 1);
}

Print Character at s

call depth 1
P_r ("Now")
returns 2
What is s? x4000

call depth 2
P_r ("ow")
returns What is stored at x4000?

'N'

Rest (2)

R5,R6

What is rest? 2

Adding 1, we obtain 3.

Return (rest + 1)
Write 3 into Return Value Slot

- Call depth 1: \texttt{p_r ("Now")}
  - Returns 3

- Call depth 2: \texttt{p_r ("ow")}
  - Returns 2

- Call depth 3: \texttt{p_r ("w")}
  - Returns

- Call depth 4: \texttt{p_r ("n")}
  - Returns 0

---

Tear Down Stack Frame

- Call depth 1: \texttt{p_r ("Now")}
  - Returns 3

- Call depth 2: \texttt{p_r ("ow")}
  - Returns 2
    - Final output is \texttt{"woN"}.

- Call depth 3: \texttt{p_r ("w")}
  - Returns
    - Return value is 3.
    - We've won!

- Call depth 4: \texttt{p_r ("n")}
  - Returns 0
    - \texttt{R5,R6→}
    - \texttt{s (x4000) →}

---

Final output is \texttt{"woN"}.
Return value is 3.
We've won!
(Sorry.)