Recursion

Matthew Mussomele

Recursion

Recursion is roughly defined as some action that happens over and over again.

In the context of C++ (and programming languages in general) recursion is defined as a function that calls itself.

It is closely linked to iteration (for and while loops).

It is a very powerful tool.

Elements of Recursion

Every recursive function must have 2 elements.

The first is the base case, and the second is the recursive call.

The base case is some set of inputs where no recursive call is made, marking the end of the recursion.

The recursive call is the part of the function that calls itself again.

Recursion Example

Below you can see a recursive implementation of the factorial function:

Base Case

Recursive Call

```
int32_t factorial(int32_t n) {
    if (n <= 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

Recursion

Recursion can be more or less intuitive than iteration, depending on the problem being solved.

For example, the fibonacci sequence is more naturally implemented recursively (although the iterative version is much faster).

Just because doing something one way or the other seems more "natural" or intuitive doesn't mean that that is the best way.

Recursive fibonacci takes time proportional to ϕ^N while the iterative solution takes time proportional to N (much faster!)

Recursion vs Iteration

Recursion	Iteration
Recursion breaks problems into many smaller subproblems and combining the results.	Keep repeating some action until the task at hand is completed.
Often shorter and more elegant (mathematicians prefer recursion).	Often is more easily understood that the recursive solution, appeals to programmers without an education in CS.
Takes more stack space (a.k.a. more function calls).	

Converting Recursion To Iteration

It is possible to convert every recursive function to an iterative one and vice versa. (See Turing Completeness)

It can be very difficult to do so in some situations.

But for many simple functions, the largest difference between the two are that the iteration state is passed as an argument in recursion.

Simply add a variable to keep track of the state and transform the recursive call into a loop to change recursion to iteration is many cases.

Converting Recursion To Iteration

Below are both the recursive and iterative versions of factorial:

```
int32_t factorial(int32_t n) {
    if (n <= 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

In the recursive version, the recursion takes care of tracking the state and simulating the loop.

Loop

State Variable

```
int32_t factorial(int32_t n) {
    int32_t n_factorial = 1;
    while (n > 1) {
        n_factorial *= n--;
    }
    return n_factorial;
}
```

Dangers of Recursion

Recursion can cause your program to crash if you do it improperly. Observe another version of factorial below:

```
int32_t factorial(int32_t n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

How might this crash a program?

Types of Recursion

There are a number of different types of recursion.

They are separated by the way in which you make recursive calls.

1. Tail-end Recursion
2. Linear Recursion
3. Tree Recursion
4. Mutual Recursion

Tail-end Recursion

Tail end recursion is very similar to iteration in nature.

It differs from other types of recursion in that the state is passed as an additional variable and the recursive call stands alone.

```c
int32_t factorial(int32_t n) {
    return tail_end_factorial(n, 1);
}

int32_t tail_end_factorial(int32_t n, int32_t result) {
    if (n <= 1) {
        return result;
    } else {
        return tail_end_factorial(n - 1, result * n);
    }
}
```

State Variable

Lone Recursive Call

Linear Recursion

Linear recursion is the most common kind of recursion.

It involves a base case and a single recursive call often paired with some kind of other expression.

The first factorial was an example of this.

```
int32_t factorial(int32_t n) {
    if (n <= 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

Expression

Recursive Call

Tail-end recursion is a special case of linear recursion.

Tree Recursion

Tree recursion occurs when there are multiple recursive calls inside of a single function.

More than one must be called on a given run of the function to count as tree recursive.

Not Tree Recursive

```
int32_t mystery(int32_t n) {
    if (n % 2) {
        mystery(n - 1);
    } else {
        mystery(n - 2);
    }
}
```

Tree Recursive

```
int32_t fibonacci(int32_t n) {
    if (n <= 1) {
        return n;
    } else {
        return fibonacci(n - 1) + fibonacci(n - 2);
    }
}
```

Tree recursive often takes much longer because of this.

Mutual Recursion

Mutual Recursion occurs when two or more functions continuously call each other back and forth.

Like a game of ping pong.

```
int32_t ping(int32_t n) {
    return pong(n + 1);
}
```

```
int32_t pong(int32_t n) {
    return ping(n - 1);
}
```

Recursive Call Order

When a function encounters a recursive call, it completely evaluates it before moving on the the rest of the function.

This behavior is exactly the same as if we were not using recursion.

This means that in tree recursion, the first recursive call made is completely evaluated before the second even starts.

Recursive Call Stacks

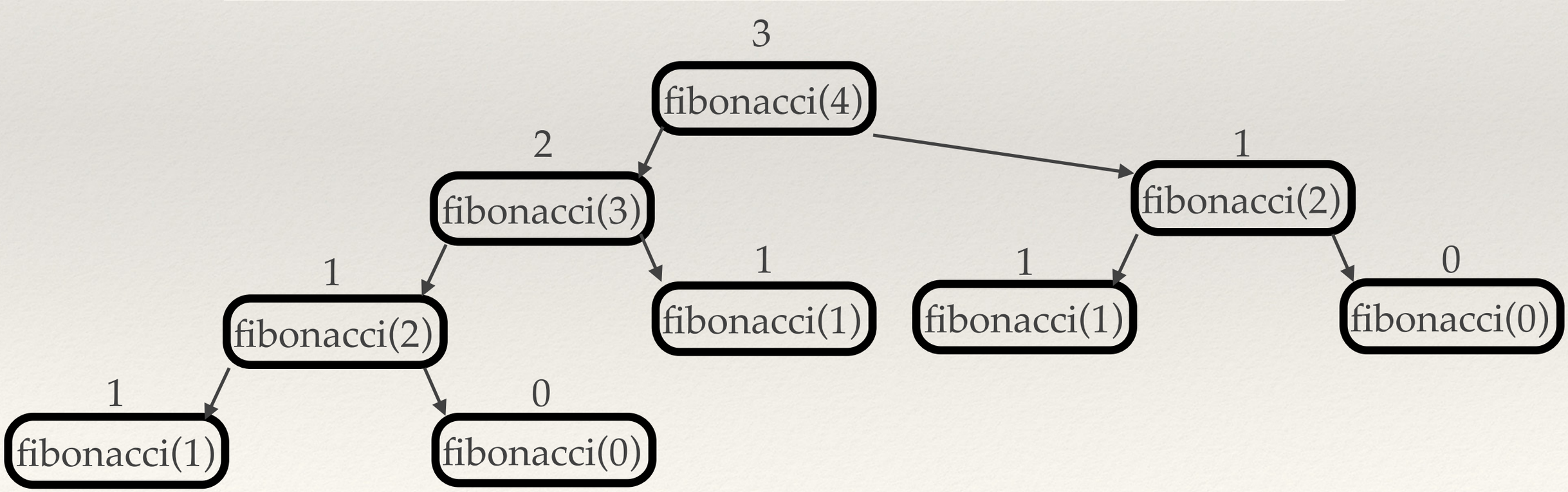
Lets look at how recursion is evaluated by the computer.

Consider the fibonacci function from earlier:

```
int32_t fibonacci(int32_t n) {
    if (n <= 1) {
        return n;
    } else {
        return fibonacci(n - 1) + fibonacci(n - 2);
    }
}
```

Recursive Call Stacks

```
int32_t fibonacci(int32_t n) {
    if (n <= 1) {
        return n;
    } else {
        return fibonacci(n - 1) + fibonacci(n - 2);
    }
}
```



Making Tree Recursion Faster

You may have noticed on the last slide that the recursive fibonacci function calculates the same thing multiple times.

We can avoid this by using a process called memoization (also known as caching).

Memoization is the process of storing the results of function calls in case they are needed again later.

See the next slide for a faster version of recursive fibonacci

Memoization

Result Storage

```
int32_t * cache = 0;

int32_t fib_memo(int32_t n) {
    if (!cache) {
        cache = new int32_t[n];
        for (int32_t i = 0; i < n; i += 1) {
            cache[i] = 0;
        }
    }
    if (n <= 1) {
        return n;
    } else if (cache[n - 1]) {
        return cache[n - 1];
    } else {
        int32_t this_fib = fib_memo(n - 1) + fib_memo(n - 2);
        cache[n - 1] = this_fib;
        return this_fib;
    }
}
```

Create Storage If Needed

Grab Stored Value

Store Found Value

Memoization

The memorization shown on the previous slide causes significant speedup of the fibonacci function.

Observe:

```
fibonacci(46) = 1836311903
Regular fibonacci took about 12.981 seconds.
fib_memo(46) = 1836311903
Memoized fibonacci took about 0.000 seconds.
```

The memoized fibonacci is still so fast it can't be measured with 3 decimal places, but the regular fibonacci is already taking over ten seconds!