# CSC 211: Computer Programming

## Recursion

Michael Conti

Department of Computer Science and Statistics
University of Rhode Island

Spring 2022

# Recursion

- Problem solving technique in which we solve a task by reducing it to smaller tasks (of the same kind)
  - then use same approach to solve the smaller tasks
- Technically, a recursive function is one that calls itself
- General form:
  - **base case**
    - solution for a **trivial case**
    - it can be used to stop the recursion (prevents "*stack overflow*")
    - every recursive algorithm needs at least one base case
  - **recursive call(s)**
    - divide problem into **smaller instance(s)** of the **same structure**



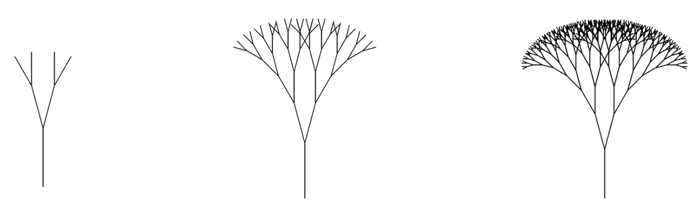
# General form

```
function() {

    if (this is the base case) {

        calculate trivial solution

    } else {

        break task into subtasks
        solve each task recursively
        combine solutions if necessary

    }
}
```

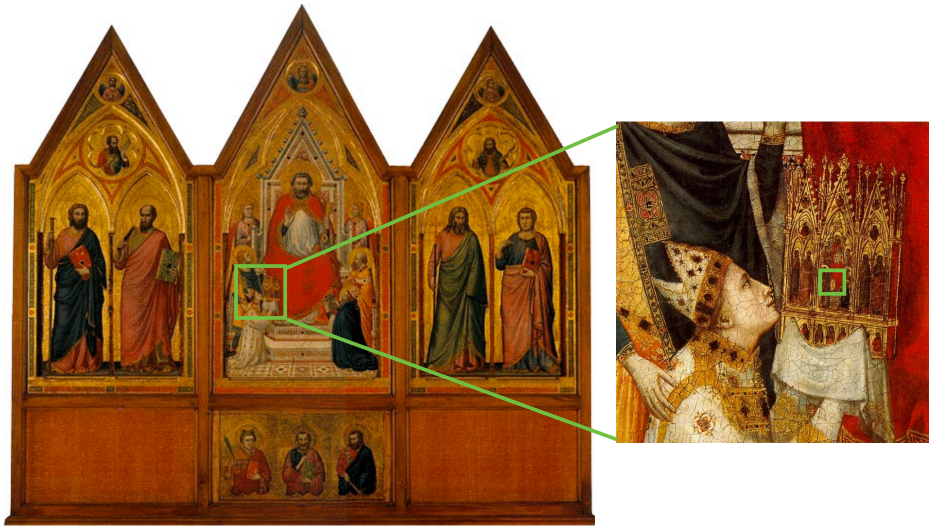


# Why recursion?

- Can we live without it?
  - yes, you can write "any program" with arrays, loops, and conditionals
- However …
  - some formulas are explicitly recursive
  - some problems exhibit a natural recursive solution



https://courses.cs.washington.edu/courses/cse120/17sp/labs/11/tree.html



The Stefaneschi Altarpiece is a triptych by the Italian medieval painter Giotto, commissioned by Cardinal Giacomo Stefaneschi to serve as an altarpiece for one of the altars of Old St. Peter's Basilica in Rome. It is now at the Pinacoteca Vaticana, Rome. Circa 1320.

https://en.wikipedia.org/wiki/Stefaneschi_Triptych



# Example: factorial

$$n! = 1 \cdot 2 \cdot \ldots \cdot n = \prod_{k=1}^{n} k$$

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ (n-1)! \cdot n & \text{if } n > 0 \end{cases}$$



# Example: factorial

- Apply the recursive definition of factorial to calculate:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ (n-1)! \cdot n & \text{if } n > 0 \end{cases}$$

- 3 !
- 5 !



# General form

```
function() {

    if (this is the base case) {

        calculate trivial solution

    } else {

        break task into subtasks
        solve each task recursively
        combine solutions if necessary

    }
}
```

## Example: factorial

```
int fact(int n) {

    // base case
    if (n < 2) {
        return 1;
    }

    // recursive call
    return fact(n-1) * n;
}
```



## Recursion call tree (tracing recursion)

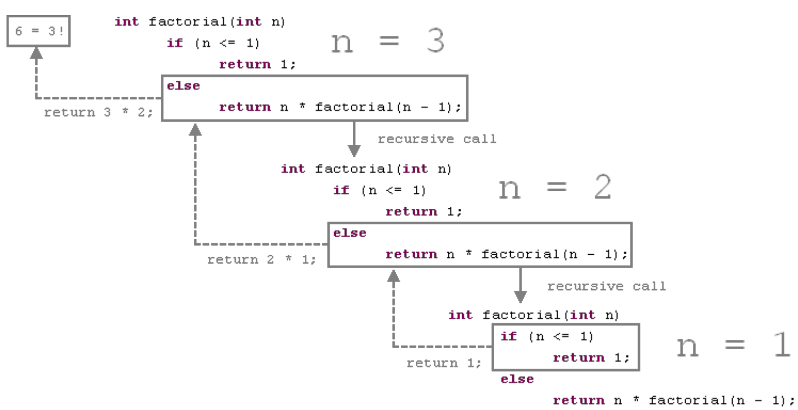
```
fact(3)
```

```
int fact(int n) {
    if (n < 2) {
        return 1;
    }
    return fact(n-1) * n;
}
```



## Example

- Factorial






## Question

- Given $f(n) = f(n-1) + 2n - 1$, what is the value of $f(3)$?

Must have base case and make progress towards base case

# Rules of the game

- Your code must have **at least one base case** for a trivial solution
  - that is, for a non-recursive solution
- Recursive calls should **make progress** towards the base case
- Your code must break a larger problem into smaller problems
  - each smaller problem should be of the same '**nature**' as the larger problem



# Example: power of a number

$$b^n = \underbrace{b \cdot b \cdot \ldots \cdot b}_{n \text{ times}}$$

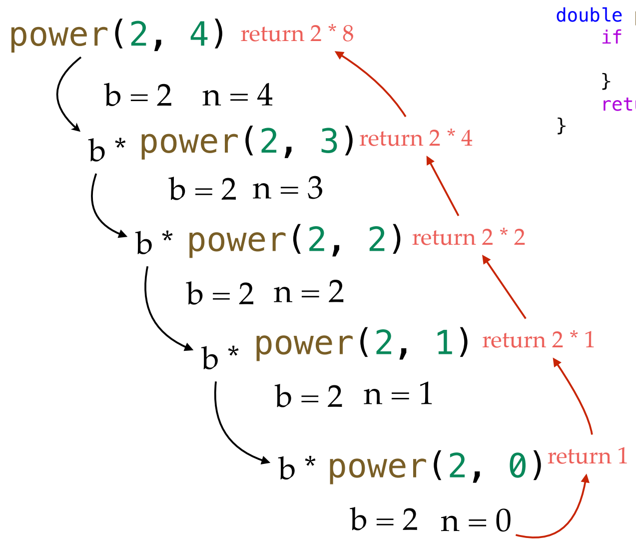
base case?

recursive case?

```
double power(double b, int n) {
    // base case
    if (n == 0) {
        return 1;
    }
    // recursive call
    return b * power(b, n-1);
}
```



# Recursion call tree (tracing recursion)




```
double power(double b, int n) {
    if (n == 0) {
        return 1;
    }
    return b * power(b, n-1);
}
```



# What is the output of `foo(1234)`?

```
int foo(int n) {
    if (n < 10) {
        return n;
    }
    int b = n % 10;
    return b + foo(n/10);
}
```

## Write a function to print this pattern?

for any n > 0

```
*
**
```

n = 2

```
*
**
***
****
```

n = 4



## Write a function to print this pattern?

for any n > 0

```
*
**
**
*
```

n = 2

```
*
**
***
****
****
***
**
*
```

n = 4



## What is the output of `mystery(7)`?

```
void mystery(unsigned int n) {
    if (n < 2) {
        std::cout << n;
    } else {
        mystery(n/2);
        std::cout << n % 2;
    }
}
```



## Indirect Recursion

```
void f2(int n);

void f1(int n) {
    if (n > 1) {
        std::cout << "1";
        f2(n - 1);
    }
}

void f2(int n) {
    std::cout << "0";
    f1(n - 1);
}
```

```
f1(1) ?

f1(2) ?

f1(4) ?

f1(7) ?

f1(10) ?
```

# Final thoughts

- Recursion is a powerful technique that solves problems by breaking them down into smaller subproblems of the same form, and applying the same strategy to solve the subproblems

- One can always write an iterative solution to a problem solved recursively
  - recursive code is often simpler to read, write, and maintain

- Not always an efficient solution (iterative counterparts are faster)
  - why not?
  - overhead

> **Overhead** is any combination of excess or indirect computation time, memory, bandwidth, or other resources that are required to perform a specific task.