Recursion
What is recursion?

- Recursion is the ability for a function to call itself.
- Why would we want to call our own function?
What is recursion?

- Recursion is the ability for a function to call itself.
- Why would we want to call our own function?
- Here's an example of a recursive function:

```java
void recursiveCount(int start) {
    println(start);
    recursiveCount(start+1);
}
```
What is recursion?

- Recursion is the ability for a function to call itself.
- Why would we want to call our own function?
- Here's an example of a recursive function:
  
  ```java
  void recursiveCount(int start) {
      println(start);
      recursiveCount(start+1);
  }
  ```

- What does it do?
Iteration vs. Recursion

- Here's a similar iterative function:

```java
void iterativeCount(int start) {
    while (true) {
        println(start++);
    }
}
```
Iteration vs. Recursion

- Here's a similar iterative function:

```java
void iterativeCount(int start) {
    while(true) {
        println(start);
        start++;
    }
}
```

- How does the above compare to:

```java
void recursiveCount(int start) {
    println(start);
    recursiveCount(start+1);
}
```
Okay, so what is it good for?
Okay, so what is it good for?

- Solving problems by breaking them into smaller pieces.
Okay, so what is it good for?

- Solving problems by breaking them into smaller pieces.
- For example how do you get 5! (five factorial)?
  - simple, that's just $5 \times 4 \times 3 \times 2 \times 1$
Okay, so what is it good for?

- Solving problems by breaking them into smaller pieces.
- For example how do you get 5! (five factorial)?
  - simple, that's just 5 x 4 x 3 x 2 x 1
- What about 30! ?
Okay, so what is it good for?

- Solving problems by breaking them into smaller pieces.

- For example how do you get 5! (five factorial)?
  - simple, that's just 5 x 4 x 3 x 2 x 1

- What about 30! ?
  - still simple, that's just 30 x 29 x … x 2 x 1
Okay, so what is it good for?

- Solving problems by breaking them into smaller pieces.
- For example how do you get 5! (five factorial)?
  - simple, that's just $5 \times 4 \times 3 \times 2 \times 1$
- What about 30! ?
  - still simple, that's just $30 \times 29 \times \ldots \times 2 \times 1$
- What about N! ?
Okay, so what is it good for?

- Solving problems by breaking them into smaller pieces.
- For example how do you get $5!$ (five factorial)?
  - simple, that's just $5 \times 4 \times 3 \times 2 \times 1$
- What about $30!$?
  - still simple, that's just $30 \times 29 \times \ldots \times 2 \times 1$
- What about $N!$?
  - Let's use a loop (iteration).
Okay, so what is it good for?

- Solving problems by breaking them into smaller pieces.

- For example how do you get 5! (five factorial)?
  - simple, that's just 5 x 4 x 3 x 2 x 1

- What about 30! ?
  - still simple, that's just 30 x 29 x ... x 2 x 1

- What about N! ?
  - Let's use a loop (iteration).
  - We could also use recursion.
Recursion – Key idea

- Keep delegating until the problem is very simple.

- In other words
  - do a simpler task each time you call yourself
  - until the task would cause no change (the base case).
N! recursive definition

- Recursive case:
  - N! is N * (N-1)!

- Base case:
  - 0! is 1

- Note: N-1 is a simpler/smaller version of the problem for factorial to act upon.
N! recursive definition

- Recursive case (let's break this down):
  - \( M = N-1; \) // make the problem smaller
  - \( \text{smallerFact} = M! \) // call factorial on the smaller problem
  - return \( N * \text{smallerFact}; \) // return result of simple multiplication.

- Base case (if \( N == 0 \)):
  - return 1;

- Note: \( N-1 \) is a simpler/smaller version of the problem for factorial to act upon.
N! recursive definition

- Recursive case:
  - N! is N * (N-1)!

- Base case:
  - 0! is 1

- So, what does this mean for 5!?

- How could we write a function to capture these 2 cases?
void setup() {
    int A = 10;
    int B = factorial(5);
    println( B );
}

int factorial(int N) {
    if (N == 1) {
        return 1;
    } else {
        int F = N * factorial(N-1);
        return F;
    }
}

Factorial – Recursive Implementation

Trace it.
Last In First Out (LIFO) Stack of Plates
1. void setup() {
2.    int A = 10;
3.    int B = factorial(5);
4.    println( B );
5. }

1. int factorial(int N) {
2.    if (N == 1) {
3.        return 1;
4.    } else {
5.        int F = N *
6.            factorial(N-1);
7.        return F;
8.    }
9. }
void setup() {
    int A = 10;
    int B = factorial(5);
    println(B);
}

int factorial(int N) {
    if (N == 1) {
        return 1;
    } else {
        int F = N * factorial(N-1);
        return F;
    }
}
void setup() {
  int A = 10;
  int B = factorial(5);
  println(B);
}

int factorial(int N) {
  if (N == 1) {
    return 1;
  } else {
    int F = N * factorial(N-1);
    return F;
  }
}

int B = factorial(5);
```java
void setup() {
    int A = 10;
    int B = factorial(5);
    println(B);
}

int factorial(int N) {
    if (N == 1) {
        return 1;
    } else {
        int F = N * factorial(N-1);
        return F;
    }
}
```
```java
int factorial(int N) {
    if (N == 1) {
        return 1;
    } else {
        int F = N * factorial(N-1);
        return F;
    }
}
```
```c
int factorial(int N=5) {
    if (N == 1) {
        return 1;
    } else {
        int F = N * factorial(N-1);
        return F;
    }
}
```
1. `void setup() {`
2. `int A = 10;`
3. `int B = factorial(5);`
4. `println( B );`
5. `}`

1. `int factorial(int N=5) {`
2. `if (N == 1) {`
3. `return 1;`
4. `}` else {`
5. `int F = N * factorial(N-1);`
6. `return F;`
7. `}`
8. `}`
1. int factorial(int N=4) {
2.   if (N == 1) {
3.     return 1;
4.   } else {
5.     int F = N * factorial(N-1);
6.     return F;
7.   }
8. }

void setup() {
1.   int A = 10;
2.   int B = factorial(5);
3.   println( B );
4. }

int factorial(int N) {
1.   if (N == 1) {
2.     return 1;
3.   } else {
4.     int F = N * factorial(N-1);
5.     return F;
6.   }
7. }
8. }

Call Stack
- setup() A=10, Line=3
- factorial() N=5, Line=5
```c
1. void setup() {
2.     int A = 10;
3.     int B = factorial(5);
4.     println( B );
5. }

1. int factorial(int N) {
2.     if (N == 1) {
3.         return 1;
4.     } else {
5.         int F = N * factorial(N-1);
6.         return F;
7.     }
8. }
```
1. int factorial(int N=4) {
2.     if (N == 1) {
3.         return 1;
4.     } else {
5.         int F = N * factorial(N-1);
6.         return F;
7.     }
8. }

1. void setup() {
2.     int A = 10;
3.     int B = factorial(5);
4.     println( B );
5. }

Call Stack
setup()  
A=10, Line=3
factorial()  
N=5, Line=5
factorial()  
N=4, Line=5
```c
1. void setup() {
2.     int A = 10;
3.     int B = factorial(5);
4.     println( B );
5. }

1. int factorial(int N=3) {
2.     if (N == 1) {
3.         return 1;
4.     } else {
5.         int F = N *
6.             factorial(N-1);
7.         return F;
8.     }
```
```cpp
int factorial(int N=3) {
    if (N == 1) {
        return 1;
    } else {
        int F = N * factorial(N-1);
        return F;
    }
}
```
```c
1. int factorial(int N=3) {
2.     if (N == 1) {
3.         return 1;
4.     } else {
5.         int F = N * 
6.         factorial(N-1);
7.         return F;
8.     }
}
```
```java
int factorial(int N=2) {
    if (N == 1) {
        return 1;
    } else {
        int F = N * factorial(N-1);
        return F;
    }
}
```
```cpp
void setup() {
    int A = 10;
    int B = factorial(5);
    println(B);
}

int factorial(int N) {
    if (N == 1) {
        return 1;
    } else {
        int F = N * factorial(N-1);
        return F;
    }
}

1. int factorial(int N=2) {
2.     if (N == 1) {
3.         return 1;
4.     } else {
5.         int F = N *
6.             factorial(N-1);
7.         return F;
8.     }
}
```
1. void setup() {
2.     int A = 10;
3.     int B = factorial(5);
4.     println(B);
5. }

1. int factorial(int N) {
2.     if (N == 1) {
3.         return 1;
4.     } else {
5.         int F = N * factorial(N-1);
6.         return F;
7.     }
8. }

1. int factorial(int N=2) {
2.     if (N == 1) {
3.         return 1;
4.     } else {
5.         int F = N * factorial(N-1);
6.         return F;
7.     }
8. }

Call Stack

- `factorial()`
  - N=2, Line=5
- `factorial()`
  - N=3, Line=5
- `factorial()`
  - N=4, Line=5
- `factorial()`
  - N=5, Line=5
- `setup()`
  - A=10, Line=3
```c
int factorial(int N=1) {
    if (N == 1) {
        return 1;
    } else {
        int F = N * factorial(N-1); 
        return F;
    }
}
```

```c
void setup() {
    int A = 10;
    int B = factorial(5);
    println(B);
}
```
1. `void setup() {`  
2. `int A = 10;`  
3. `int B = factorial(5);`  
4. `println( B );`  
5. `}`

1. `int factorial(int N) {`  
2. `if (N == 1) {`  
3. `return 1;`  
4. `} else {`  
5. `int F = N * factorial(N-1);`  
6. `return F;`  
7. `}`  
8. `}
1. `int factorial(int N=2) {`
2.     if (N == 1) {
3.         return 1;
4.     } else {
5.         int F = N * factorial(N-1);
6.         return F;
7.     }
8. }

1. `void setup() {
2.     int A = 10;
3.     int B = factorial(5);
4.     println( B );
5. }

Call Stack

- `factorial()`
  - N=3, Line=5
- `factorial()`
  - N=4, Line=5
- `factorial()`
  - N=5, Line=5
- `setup()`
  - A=10, Line=3
1. int factorial(int N=3) {
2.   if (N == 1) {
3.     return 1;
4.   } else {
5.     int F = N * 2;
6.     return F;
7.   }
8. }

1. void setup() {
2.   int A = 10;
3.   int B = factorial(5);
4.   println( B );
5. }

1. int factorial(int N) {
2.   if (N == 1) {
3.     return 1;
4.   } else {
5.     int F = N * factorial(N-1);
6.     return F;
7.   }
8. }
```c
void setup() {
    int A = 10;
    int B = factorial(5);
    println(B);
}

int factorial(int N) {
    if (N == 1) {
        return 1;
    } else {
        int F = N * factorial(N-1);
        return F;
    }
}
```

Call Stack:
- setup()
  - A=10, Line=3
- factorial()
  - N=5, Line=5
```java
int factorial(int N=5) {
    if (N == 1) {
        return 1;
    } else {
        int F = N * factorial(N-1);
        return F;
    }
}
```

```java
void setup() {
    int A = 10;
    int B = factorial(5);
    println(B);
}
```
void setup() {
    int A = 10;
    int B = factorial(5);
    println(B);
}

int factorial(int N) {
    if (N == 1) {
        return 1;
    } else {
        int F = N * factorial(N-1);
        return F;
    }
}

void setup() {
    int A = 10;
    int B = 120;
    println(B);
}
The Call Stack keeps track of …

1. all functions that are suspended, in order

2. the point in the function where execution should resume after the invoked subordinate function returns

3. a snapshot of all variables and values within the scope of the suspended function so these can be restored upon continuing execution
What happens if there is no stopping condition, or "base case"?
Recursive Drawing

- Draw a shape, then recursively draw a smaller version of the shape.

- Examples:
  - drawCircles
  - drawSquares
Creating a maze, recursively

1. Start with a rectangular region defined by its upper left and lower right corners

2. Divide the region at a random location through its more narrow dimension

3. Add an opening at a random location

4. Repeat on two rectangular subregions

Inspired by
RecursiveMaze

int N = 25;     // Grid dimension
int gsize = 20; // Grid size

int V = 1;      // Vertical constant
int H = 2;      // Horizontal constant

void setup() {
  // Setup sketch
  size(N*gsize+1, N*gsize+1);
  noLoop();
  background(255);
  stroke(0);

  // Kick off the recursive divide
  // on entire sketch window
  divide(0,0,N,N);
}

// Determine the direction for dividing
// Stop when too small.
int divDir(int r1, int c1, int r2, int c2) {
  int dr = r2 - r1;        // Deltas
  int dc = c2 - c1;
  if (dr <= 1 || dc <= 1)  // Too small
    return 0;              // No division
  else if (dr < dc)        // Flat and wide
    return V;              // Vertical division
  else                     // Tall and narrow
    return H;              // Horizontal div
}

// Return a random integer in the range
int randomInt(int min, int max) {
  return round(random(min-0.5,max+0.5));
}

// Draw a line on a grid segment
void gridLine(int r1, int c1, int r2, int c2) {
  line(r1*gsize, c1*gsize, r2*gsize, c2*gsize);
}
// Divide the region given upper left and
// lower right grid corner points

void divide(int r1, int c1, int r2, int c2) {
    int cr, rr;

    // Get divide direction (V, H or 0)
    int dir = divDir(r1, c1, r2, c2);

    // Divide in vertical direction
    if (dir == V) {
        // Wall and opening locations
        cr = randomInt(c1+1, c2-1);
        rr = randomInt(r1, r2-1);

        // Draw wall
        gridLine(cr, r1, cr, rr);
        gridLine(cr, rr+1, cr, r2);

        // Recursively divide two subregions
        divide(r1, c1, r2, cr);
        divide(r1, cr, r2, c2);
    }
    // Divide in horizontal direction
    else if (dir == H) {
        // Wall and opening locations
        cr = randomInt(c1, c2-1);
        rr = randomInt(r1+1, r2-1);

        // Draw wall
        gridLine(c1, rr, cr, rr);
        gridLine(cr+1, rr, c2, rr);

        // Recursively divide two subregions
        divide(r1, c1, rr, c2);
        divide(rr, c1, r2, c2);
    }
    // No division. We're done.
    else {
        return;
    }
}