recursion

defining a function in terms of itself
DIVIDE AND CONQUER
Special Service Division
INFORMATION FILM #3
factorial

the product of all positive integers less than or equal to a given non-negative number
factorial

the product of all positive integers less than or equal to a given non-negative number

5!
factorial
the product of all positive integers less than or equal to a given non-negative number

\[ 5! = 5 \times 4 \times 3 \times 2 \times 1 \]
Recursive Definition
$n!$
\[ n! = n \times (n - 1)! \]
\[ n! = n \times (n - 1)! \]
\[ n! = n \times (n - 1)! \]

5!
\[ n! = n \times (n - 1)! \]

\[ 5! = 5 \times 4! \]
Recursive solutions must satisfy three rules:
Recursive solutions **must** satisfy three rules:

1. Contain a **recursive case**
Recursive solutions **must** satisfy three rules:

1. Contain a **recursive case**
2. Contain a **base case**
Recursive solutions must satisfy three rules:

1. Contain a recursive case
2. Contain a base case
3. Must make progress toward the base case
Recursive solutions **must** satisfy three rules:

1. Contain a recursive case
2. Contain a base case
3. Must make progress toward the base case
Every recursive function has an equivalent iterative solution.
Every iterative function has an equivalent recursive solution.
Every iterative function has an equivalent recursive solution.

Some problems can be easier to solve one way than the other.
input \rightarrow function \rightarrow output
How many students?
How many students?

iterative vs. distributed counting
int factorial(int n)
{
    int i, product = 1;

    /* computes n*n-1... */
    for (i=n; i>1; i=i-1)
    {
        product = product * i;
    }

    /* the value returned */
    return (product);
}
int factorial (int n) {
    int product;

    if (n == 0)
        product = 1;
    else
        product = n * factorial(n-1);

    return (product);
}
int factorial (int n)
{
    int product;

    if (n == 0)
        product = 1;
    else
        product = n * factorial(n-1);

    return (product);
}
Recursive Version

```c
int factorial (int n) {
    int product;

    if (n == 0)
        product = 1;
    else
        product = n * factorial(n-1);

    return (product);
}
```

What is the recursive case?
int factorial (int n) {
    int product;

    if (n == 0)
        product = 1;
    else
        product = n * factorial(n-1);

    return (product);
}
int factorial (int n)
{
    int product;

    if (n == 0)
        product = 1;
    else
        product = n * factorial(n-1);

    return (product);
}
int factorial (int n) {
    int product;

    if (n == 0)
        product = 1;
    else
        product = n * factorial(n-1);

    return (product);
}
int factorial (int n)
{
    int product;

    if (n == 0)
        product = 1;
    else
        product = n * factorial(n-1);

    return (product);
}
Function Call Trace
factorial(5)
factorial(5)

factorial(4)
factorial(5)
factorial(4)
factorial(3)
factorial(5)
factorial(4)
factorial(3)
factorial(2)
factorial(5)
factorial(4)
factorial(3)
factorial(2)
factorial(1)
factorial(5)
factorial(4)
factorial(3)
factorial(2)
factorial(1)
return 1

Function Call Trace
factorial(5)
factorial(4)
factorial(3)
factorial(2)
factorial(1)
return 1

base case
factorial(5)
factorial(4)
factorial(3)
factorial(2)
return 1
return 2*1 = 2
factorial(5)
factorial(4)
factorial(3)
factorial(2)
return 1
return 2*1 = 2
return 3*2 = 6
factorial(5)
factorial(4)
factorial(3)
factorial(2)
return 1
return 2*1 = 2
return 3*2 = 6
return 4*6 = 24
factorial(5)
factorial(4)
factorial(3)
factorial(2)
return 1
return 2*1 = 2
return 3*2 = 6
return 4*6 = 24
return 5*24 = 120
factorial(5)
factorial(4)
factorial(3)
factorial(2)

Computing running product starting from base case

return 1
return 2*1 = 2
return 3*2 = 6
return 4*6 = 24
return 5*24 = 120
int multiply(int m, int n) {
    int answer;

    if (n == 1)
        answer = m;
    else
        answer = m + multiply(m, n - 1);

    return answer;
}
int multiply(int m, int n)
{
    int answer;

    if (n == 1)
        answer = m;
    else
        answer = m + multiply(m, n - 1);

    return answer;
}
void f(int m, int n) {
    int answer;

    if (n == 1)
        answer = m;
    else
        answer = m + f(m, n - 1);

    return answer;
}

What is the recursive case?
int multiply(int m, int n)
{
    int answer;

    if (n == 1)
        answer = m;
    else
        answer = m + multiply(m, n - 1);

    return answer;
}

What is the recursive case?
```c
int multiply(int m, int n)
{
    int answer;

    if (n == 1)
        answer = m;
    else
        answer = m + multiply(m, n - 1);

    return answer;
}
```
```c
int multiply(int m, int n)
{
    int answer;

    if (n == 1)
        answer = m;
    else
        answer = m + multiply(m, n - 1);

    return answer;
}
```

What is the base case?
int multiply(int m, int n) {
    int answer;

    if (n == 1)
        answer = m;

    else
        answer = m + multiply(m, n - 1);

    return answer;
}

What is the base case?
int multiply(int m, int n)
{
    int answer;

    if (n == 1)
        answer = m;
    else
        answer = m + multiply(m, n - 1);

    return answer;
}
Write-out the recursive multiplication function call trace for \( m = 3 \) and \( n = 4 \).
Fibonacci number:

\[ F_n = F_{n-1} + F_{n-2} \quad n > 2 \]
Fibonacci number:

\[ F_n = F_{n-1} + F_{n-2} \quad n > 2 \]

\[ F_1 = 1 \]

\[ F_2 = 1 \]
Fibonacci number:

\[ F_n = F_{n-1} + F_{n-2} \quad n > 2 \]

\[ F_1 = 1 \]

\[ F_2 = 1 \]

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144
Fibonacci number:

\[ F_n = F_{n-1} + F_{n-2} \quad n > 2 \]

\[ F_1 = 1 \]

\[ F_2 = 1 \]

\[ F_{12} \]

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144
Fibonacci number:

\[ F_n = F_{n-1} + F_{n-2} \quad n > 2 \]

\[ F_1 = 1 \]

\[ F_2 = 1 \]

\[ 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144 \]
Fibonacci number:

\[ F_n = F_{n-1} + F_{n-2} \quad n > 2 \]

\[ F_1 = 1 \]

\[ F_2 = 1 \]

\[ F_{10} = 55, \quad F_{12} = 89, \quad 144 \]
Fibonacci number:

\[ F_n = F_{n-1} + F_{n-2} \quad n > 2 \]

\[ F_1 = 1 \]

\[ F_2 = 1 \]

1, 1, 2, 3, 5, 8, 13, 21, 34, \ \underline{55}, \ \underline{89}, \ \underline{144}\ 

\[ + \quad = \]
Fibonacci number:

\[ F_n = F_{n-1} + F_{n-2} \quad n > 2 \]

\[ F_1 = 1 \]

\[ F_2 = 1 \]

\[ F_{10} \quad F_{11} \quad F_{12} \]

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144

+ =
int fibonacci (int N) {
    int k1, k2, k3;
    k1 = k2 = k3 = 1;

    for (int j = 3; j <= N; j++) {
        k3 = k1 + k2;
        k1 = k2;
        k2 = k3;
    }

    return k3;
}
int fibonacci(int N)
{
    if ( (N == 1) || (N == 2) )
    {
        return 1;
    }
    else
    {
        return ( fibonacci(N-1) + fibonacci(N-2) );
    }
}
int fibonacci(int N)
{
    if ( (N == 1) || (N == 2) )
    {
        return 1;
    }
    else
    {
        return
        ( fibonacci(N-1) + fibonacci(N-2) );
    }
}
int fibonacci(int N) {
    if ( (N == 1) || (N == 2) )
    {
        return 1;
    }
    else
    {
        return (fibonacci(N-1) + fibonacci(N-2));
    }
}
int fibonacci(int N) {
    if ((N == 1) || (N == 2)) {
        return 1;
    } else {
        return (fibonacci(N-1) + fibonacci(N-2));
    }
}
int fibonacci(int N) {
    if ( (N == 1) || (N == 2) ) {
        return 1;
    } else {
        return (fibonacci(N-1) + fibonacci(N-2));
    }
}
int fibonacci(int N) {
    if ( (N == 1) || (N == 2) ) {
        return 1;
    }
    else {
        return (fibonacci(N-1) + fibonacci(N-2));
    }
}
```c
int fibonacci(int N) {
    if ( (N == 1) || (N == 2) ) {
        return 1;
    } else {
        return (fibonacci(N-1) + fibonacci(N-2));
    }
}
```

**Recursive Version**

What is the base case?
int fibonacci(int N)
{
    if ( (N == 1) || (N == 2) )
    {
        return 1;
    }
    else
    {
        return ( fibonacci(N-1) + fibonacci(N-2) );
    }
}
Only one disk can move at a time
Only one disk can move at a time

Each move takes the upper disk from one stack and places it on top of another stack
Only one disk can move at a time

Each move takes the upper disk from one stack and places it on top of another stack

No disk may be placed on top of a smaller disk
Tower of Hanoi
7 Rings
Tower of Hanoi
7 Rings
void TowersOfHanoi(int n, char a, char b, char c)
{
    if(n==1)
        printf("\nMoved from %c to %c",a,c);
    else
    {
        TowersOfHanoi(n-1,a,c,b);
        TowersOfHanoi(1,a," ",c);
        TowersOfHanoi(n-1,b,a,c);
    }
}
re-curs-ion (ri-kur'-zhin)
n.
   1. see recursion