Recursion

To understand recursion, you must first understand recursion

Recursion

- A recursive function is a function that is defined in terms of itself.
- Every recursive definition must have a base case that is not recursive.
  - The non-recursive nature of the base case allows us to then solve previous recursive steps.
- There can be more than one base case.
Factorial

- \( n! = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1 \) for \( n > 0 \)
  \[= 1 \text{ for } n = 0\]
- But, since \((n-1)! = (n-1) \times (n-2) \times 2 \times 1\), we can use recursion to define the factorial function:
  \( n! = n \times (n-1)! \) for \( n > 0 \)
  \[= 1 \text{ for } n = 0 \text{ (base case)}\]
- Example:
  \[4! = 4 \times 3! = 4 \times (3 \times 2!) = 4 \times (3 \times (2 \times 1)) = 4 \times (3 \times (2 \times 1 \times 1)) = 4 \times 6 = 24\]

Factorial in Java

```java
public static int factorial(int n) {
    // Precondition: n >= 0
    int result;
    if (n == 0)
        result = 1;
    else
        result = n * factorial(n-1);
    return result;
}
```
Improved Factorial

```java
public static int factorial(int n) {
    if (n < 0)
        throw new IllegalArgumentException();
    int result;
    if (n == 0)
        result = 1;
    else
        result = n * factorial(n-1);
    return result;
}
```

Another Factorial

```java
public static int factorial(int n) {
    if (n < 0)
        throw new IllegalArgumentException();
    return fact(n, 1);
}

private static int fact(int n, int total) {
    if (n == 0)
        return total;
    return fact(n-1, n*total); // tail recursive
}
```
Tail Recursion

A tail recursive method can always be converted into an iterative one. Which is more efficient?

```java
public static int factorial(int n) {
    // Precondition: n >= 0
    int result = 1;
    for (int k = 1; k <= n; k++)
        result *= k;
    return result;
}
```

Recursion and the Run-Time Stack

```java
... int x = factorial(4);
...
```

```java
public static int factorial(int n) {
    // Precondition: n >= 0
    int result;
    if (n == 0)
        result = 1;
    else
        result = n * factorial(n-1);
    return result;
}
```
Fibonacci Numbers

Fibonacci Numbers in Java

\[ f_{ib_n} = f_{ib_{n-1}} + f_{ib_{n-2}} \quad \text{for } n > 2 \]
\[ f_{ib_2} = 1 \]
\[ f_{ib_1} = 1 \]

```java
public static int fib(int n) {
    // Precondition: n > 0
    if (n <= 2)
        return 1;
    else
        return fib(n-1) + fib(n-2);
}
```

Fibonacci Numbers

fib(6)

fib(5)

fib(4)

fib(4)  fib(3)

fib(3)  fib(2)

fib(2)  fib(2)

fib(1)  fib(1)

fib(2)  fib(1)

fib(2)  fib(1)

fib(2)  fib(1)

fib(2)  fib(1)

fib(2)  fib(1)

fib(2)  fib(1)

fib(2) = 1
fib(1) = 1
fib(n) = fib(n-1) + fib(n-2), n > 2

Fibonacci Numbers

8

5  3

3  2  2  1

2  1  1  1  1

1  1  1  1  1

fib(1) = 1
fib(2) = 1
fib(n) = fib(n-1) + fib(n-2), n > 2
Fibonacci Numbers Iteratively

```java
public static int fib(int n) {
    // Precondition: n > 0
    int fibprev2 = 0;
    int fibprev = 1;
    int fibcurr = 1;
    for (int i = 3; i <= n; i++) {
        fibprev2 = fibprev;
        fibprev = fibcurr;
        fibcurr = fibprev + fibprev2;
    }
    return fibcurr;
}
```

Order of complexity:

- Iterative: $O(n)$
- Recursive: $O(2^n)$

If $n$ is too high, this can result in StackOverflowError.

Greatest Common Divisor

```java
public static int gcd(int m, int n) {
    // Precondition: m > 0, n > 0
    if (m % n == 0)
        return n;
    else
        return gcd(n, m % n);
}
```
String length

public static int length(String s) {
    // Precondition: s != null
    if (s == null)
        return 0;
    else
        return ___________________________;
}

Linear Search

private static int search(int[] a, int target, int index) {
    // Search array a for target starting at index
    if (______________________________)
        return index;
    else if (______________________________)
        return -1;
    else
        return ___________________________;
Use of wrapper method

```java
public static int search(int[] a, int target)
{
    return search(a, target, 0); // prev. slide
}
```

User calls this method to search entire array. This method calls the recursive method to start the search with index 0.

Exponentiation ($x^n$)

$x^n = \underbrace{x \times x \times \ldots x}_{n-1} = x \times x^{n-1}$, for $n > 0$.

```java
public static double power(double x, int n)
{
    // Precondition: n >= 0
}
What's wrong?

We can also say (recursively) that $x^n = x^{n/2} \cdot x^{n/2}$.

What's wrong with the following recursive step for exponentiation?

```java
return power(x, n/2) * power(x, n/2);
```

Linked Lists Recursively

Assume we have a singly-linked list as defined below:

```java
public class SinglyLinkedList {
    private Node<E> head;

    private static class Node<E> {
        ...
    }
}
```
Finding the size of the list

```java
public int size() { // wrapper method
    size(head);
}
private int size(Node<E> nodeRef) {
    // Find size of list starting at nodeRef
    if (nodeRef == null)
        return 0;
    else
        return 1 + size(nodeRef.next);
}
```

Adding a new node to the end of the list

```java
public void add(E element) { // wrapper method
    if (head == null)
        head = new Node<E>(element);
    else
        add(head, element);
}
private void add(Node<E> nodeRef, E element) {
    if (nodeRef.next == null)
        nodeRef.next = new Node<E>(element);
    else
        add(nodeRef.next, element);
}
```
What's wrong?

public void add(E element) { // wrapper method
    add(head, element); // Add element to the end of the list that starts
} // at the node referenced by nodeRef

private void add(Node<E> nodeRef, E element) {
    if (nodeRef == null)
        nodeRef = new Node<E>(element);
    else
        add(nodeRef.next, element);
}

Counting number of matches

public int count(E element) { // wrapper method
    return count(head, element);
} // wrapper method

private int count(Node<E> nodeRef, E element) {
    // Returns number of matches of element in linked list
    // starting from node referenced by nodeRef
}
Towers of Hanoi

- A puzzle invented by French mathematician Edouard Lucas in 1883.
- At a monastery far away, monks were led to a courtyard with three pegs and 64 discs stacked on one peg in size order.
  - Monks are only allowed to move one disc at a time from one peg to another.
  - Monks may not put a larger disc on top of a smaller disc at any time.
- The goal of the monks was to move all 64 discs from the leftmost peg to the rightmost peg.
- According to the legend, the world would end when the monks finished their work.

Recursive Solution

Move N discs from peg A to peg C (Let B represent the extra peg.)

- a. If N > 1, move N-1 discs from peg A to peg B.
- b. Move 1 disc from peg A to peg C.
- c. If N > 1, move N-1 discs from peg B to peg C.
Backtracking

- Algorithmic technique to search a large problem space for a solution.
- At each point in the search, we have a number of choices.
- As we pick a choice and proceed on, we may hit a "dead end".
- Backtracking involves "backing up" to the most recent choice and choosing another possible choice to follow.

Finding a path in a maze

- Let a maze be represented as an $m \times n$ array with two colors.
  - All cells that are part of the maze are painted in one color (GREEN).
  - All cells that are not part of the maze are painted in another color (RED).
- The entry point in the maze is (0,0).
- The exit point in the maze is (m-1, n-1).
Finding a path in a maze

Is there a path from cell \(x,y\) to the exit?
Initially, \(x = 0\) and \(y = 0\).

If there is a path, mark each node along the path in blue.
Finding a path in a maze

### Base Cases

Is there a path from \((x,y)\) to \((m-1, n-1)\)?
- If \((x,y)\) is outside the maze boundaries, answer NO.
- If \((x,y)\) is a RED cell, answer NO.
- If \((x,y)\) has already been visited, answer NO.
- If \((x,y)\) is \((m-1, n-1)\),
  color this cell in blue and answer YES.

How do we know if a cell has already been visited?
- Color the cell in yellow.

### Recursive Step

Is there a path from \((x,y)\) to \((m-1, n-1)\)?
If none of the base cases apply...
- Color cell \((x,y)\) in blue.
- For each neighbor of \((x,y)\),
  - If there is a path from the neighbor to \((m-1,n-1)\),
    answer YES.
- Otherwise, recolor cell \((x,y)\) with yellow (i.e. visited but not part of the path) and answer NO.