Researchers at Microsoft, UW and U Toronto have come up with a technique to interact with a computer by flexing muscles (sensor electrodes on forearm)
You already know it!

n! = n \times (n - 1)!

A king is a son of a king

Family Tree of Rabbits
**Definition**

- **Recursion**: (noun) See recursion. 😊

- An algorithmic technique where a function, in order to accomplish a task, calls itself with some part of the task.

- Recursive solutions involve two major parts:
  - **Base case(s)**, the problem is simple enough to be solved directly
  - **Recursive case(s)**. A recursive case has three components:
    - Divide the problem into one or more simpler or smaller parts
    - Invoke the function (recursively) on each part, and
    - Combine the solutions of the parts into a solution for the problem.

- Depending on the problem, any of these may be trivial or complex.

www.nist.gov/dads/HTML/recursion.html
Linear Functional Pattern

- **Functional programming**
  - It’s all about the reporter **return value**
  - There are **no side-effects**

- **Recursion**\( (\text{arg}) \) 
  \[
  \text{if}(\text{base\_case\_test}) \ \{ \\
  \quad \text{return}(\text{base\_case\_answer}) \\
  \} \ \text{else} \ \{ \\
  \quad \text{return}(\text{Combiner}(\text{SomePart}(\text{arg}), \text{Recursion}(\text{Rest}(\text{arg})))) \\
  \} \\
  \]

- **Base case(s)**
- **Recursive case(s)**
  - Divide
  - Invoke
  - Combine
Linear Functional Example: $n!$

### Factorial($n$) = $n!$

**Inductive definition:**
- $n! = 1$, $n = 0$
- $n! = n * (n-1)!$, $n > 0$

### What are...

- **base_case_test**
  - $n == 0$
- **base_case_answer**
  - 1
- **SomePart**($arg$)
  - $n$
- **Rest**($arg$)
  - $n-1$
- **Combiner:** *

Let's now trace...

```python
def Recursion(arg):
    if(base_case_test):
        return(base_case_answer)
    else:
        return(Combiner(SomePart(arg),
                        Recursion(Rest(arg)))))
```

```python
def Factorial(n):
    if(n == 0):
        return(1)
    else:
        return(n * Factorial(n-1))
```

Let's now trace...

```
Factorial n
if n = 0
set answer to 1
else
set answer to n * factorial n - 1
report answer
```

---

**Notes:**
- $n! = \prod_{k=1}^{n} k$ for all $n \in \mathbb{N} \geq 0$.
Non-linear Functional Example: \textit{Fib}

<table>
<thead>
<tr>
<th>n</th>
<th>(F(n))</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

\begin{itemize}
  \item \textbf{Inductive definition:}
    \[
    F(n) = \begin{cases} 
    0 & \text{if } n = 0; \\
    1 & \text{if } n = 1; \\
    F(n-1) + F(n-2) & \text{if } n > 1.
    \end{cases}
    \]
  \item \textbf{What are…}
    \begin{itemize}
      \item \texttt{base\_case\_test} \\
      \hspace{1em} \texttt{n} <= 1 \\
      \item \texttt{base\_case\_answer} \\
      \hspace{1em} \texttt{n} \\
      \item \texttt{SomePart(arg)} \\
      \hspace{1em} \texttt{0} \\
      \item \texttt{Rest(arg)} \\
      \hspace{1em} \texttt{n-1} \text{ and } \texttt{n-2} \\
      \item \texttt{Combiner} \\
      \hspace{1em} +
    \end{itemize}
\end{itemize}

\begin{verbatim}
Recursion(arg) {
  if(base_case_test) {
    return(base_case_answer)
  } else {
    return(Combiner(SomePart(arg),
                    Recursion(Rest(arg)))))
  }
}
Recursion(arg) {
  if(base_case_test) {
    return(base_case_answer);
  } else {
    return(Combiner(SomePart(arg),
                    Recursion(Rest1(arg)),
                    Recursion(Rest2(arg)),
                    ...
                    Recursion(Restn(arg)))
  }
}
Fib(n) {
  if(n <= 1) {
    return(n)
  } else {
    return(Fib(n-1)+Fib(n-2))
  }
}
\end{verbatim}

Let’s now: trace… (gif from Ybungalobill@wikimedia)

Leonardo de Pisa aka, Fibonacci
Authoring: Trust the Recursion!

- When authoring recursive code:
  - The base is usually easy: “when to stop?”
  - In the recursive step
    - How can we break the problem down into two:
      - A piece I can handle right now
      - The answer from a smaller piece of the problem
    - Assume your self-call does the right thing on a smaller piece of the problem
    - How to combine parts to get the overall answer?

- Practice will make it easier to see idea
Now you try one...

- Want to park unit-length cars on a block 10 units wide.
- In the ideal case, you can get 10 cars in.
  - Assume no wiggling needed, they just drop in
- Assuming cars arrive & park randomly on the open spaces, how many cars can park on avg?
- With a partner, write \( \text{park}(\text{room}) \rightarrow \# \text{ of cars} \)
  - Answer will be \( \text{park}(10) \)

E.g.,
- Given: \( \text{place}(\text{room}) \) randomly places rear bumper on a place from 0 to \( \text{room}-1 \) and returns location. Assumes \( \text{room} > 1 \)!

\[ \text{place}(10) \rightarrow 2 \]
10^7 trials; avg = 7.2235337

```java
park(room) {
    if (room < 1) {
        return 0;
    } else {
        rear = place(room);
        return (park(rear) + 1 +
            park(room - (rear + 1)));
    }
}
```

Number of simulations:
- 0: 5681623
- 1: 3098303
- 2: 1093706
- 3: 120869
- 4: 0
- 5: 0
- 6: 0
- 7: 0
- 8: 0
- 9: 0
- 10: 0

Number of cars parked:
- 0: 5499
- 1: 1093706
- 2: 3098303
- 3: 120869
- 4: 0
- 5: 0
- 6: 0
- 7: 0
- 8: 0
- 9: 0
- 10: 0
Fractal Beauty in Nature

- Fractals are self-similar objects
  - They appear in nature: rivers, coastlines, mountains, snow
- They are perfect for describing with recursion
  - Same ideas apply: base case + recursive case
  - Tip: look at $n=0$ and $n=1$ case; $n=\infty$ is often hard to decrypt
- How to write (pseudo)code for the Sierpinski Square:
  - `SierpinskiSquare(L,R,U,D,n)` given `DrawRectangle(L,R,U,D)` and `OneThird(from,to)`
Sierpinski Square

SierpinskiSquare(L,R,U,D,n) {
    if(n == 0) {
        DrawRectangle(L,R,U,D)
    } else { // We shorten OneThird to OT here...
        SierpinskiSquare(L,OT(L,R),U,OT(U,D),n-1) // NW
        SierpinskiSquare(OT(R,L),R,U,OT(U,D),n-1) // NE
        SierpinskiSquare(OT(L,R),OT(R,L),OT(U,D),OT(D,U),n-1)
        SierpinskiSquare(L,OT(L,R),OT(D,U),D,n-1) // SW
        SierpinskiSquare(OT(R,L),R,OT(D,U),D,n-1) // SE
    }
}

- This is procedural recursion -- purpose is side-effect, not return value
  - Sometimes we want a side-effect AND a return value...
Conclusion

- Many flavors, patterns
  - Functional
  - Procedural
- Inductive definitions lead naturally to recursion
- Recursion simpler code
  - Fractals scream for it
- Thinking recursively
  - Breaking problem down by trusting recursion, building off of that