Announcements
Announcements

• Homework 2 due Tuesday at 11:59pm
Announcements

• Homework 2 due Tuesday at 11:59pm
• Project 1 due Thursday at 11:59pm
Announcements

• Homework 2 due Tuesday at 11:59pm
• Project 1 due Thursday at 11:59pm
  ▪ Extra debugging office hours in Soda 405: Tuesday 6–8, Wednesday 6–7, Thursday 5–7
Announcements

• Homework 2 due Tuesday at 11:59pm
• Project 1 due Thursday at 11:59pm
  • Extra debugging office hours in Soda 405: Tuesday 6–8, Wednesday 6–7, Thursday 5–7
  • Readers hold these office hours; they are the ones who give you composition scores!
Announcements

• Homework 2 due Tuesday at 11:59pm
• Project 1 due Thursday at 11:59pm
  ▪ Extra debugging office hours in Soda 405: Tuesday 6–8, Wednesday 6–7, Thursday 5–7
  ▪ Readers hold these office hours; they are the ones who give you composition scores!
• Optional guerrilla section Monday 6pm–8pm, meeting outside of Soda 310
Announcements

- Homework 2 due Tuesday at 11:59pm
- Project 1 due Thursday at 11:59pm
  - Extra debugging office hours in Soda 405: Tuesday 6–8, Wednesday 6–7, Thursday 5–7
  - Readers hold these office hours; they are the ones who give you composition scores!
- Optional guerrilla section Monday 6pm–8pm, meeting outside of Soda 310
- Midterm 1 is next Monday 9/23 from 7pm to 9pm in various locations across campus
Announcements

• Homework 2 due Tuesday at 11:59pm
• Project 1 due Thursday at 11:59pm
  ▪ Extra debugging office hours in Soda 405: Tuesday 6–8, Wednesday 6–7, Thursday 5–7
  ▪ Readers hold these office hours; they are the ones who give you composition scores!
• Optional guerrilla section Monday 6pm–8pm, meeting outside of Soda 310
• Midterm 1 is next Monday 9/23 from 7pm to 9pm in various locations across campus
  ▪ Closed book, paper-based exam.
Announcements

• Homework 2 due Tuesday at 11:59pm
• Project 1 due Thursday at 11:59pm
  ▪ Extra debugging office hours in Soda 405: Tuesday 6–8, Wednesday 6–7, Thursday 5–7
  ▪ Readers hold these office hours; they are the ones who give you composition scores!
• Optional guerrilla section Monday 6pm–8pm, meeting outside of Soda 310
• Midterm 1 is next Monday 9/23 from 7pm to 9pm in various locations across campus
  ▪ Closed book, paper-based exam.
  ▪ You may bring one hand-written page of notes that you created (front & back).
Announcements

• Homework 2 due Tuesday at 11:59pm
• Project 1 due Thursday at 11:59pm
  ▪ Extra debugging office hours in Soda 405: Tuesday 6-8, Wednesday 6-7, Thursday 5-7
  ▪ Readers hold these office hours; they are the ones who give you composition scores!
• Optional guerrilla section Monday 6pm-8pm, meeting outside of Soda 310
• Midterm 1 is next Monday 9/23 from 7pm to 9pm in various locations across campus
  ▪ Closed book, paper-based exam.
  ▪ You may bring one hand-written page of notes that you created (front & back).
  ▪ You will have a study guide attached to your exam.
Announcements

• Homework 2 due Tuesday at 11:59pm
• Project 1 due Thursday at 11:59pm
  ▪ Extra debugging office hours in Soda 405: Tuesday 6-8, Wednesday 6-7, Thursday 5-7
  ▪ Readers hold these office hours; they are the ones who give you composition scores!
• Optional guerrilla section Monday 6pm-8pm, meeting outside of Soda 310
• Midterm 1 is next Monday 9/23 from 7pm to 9pm in various locations across campus
  ▪ Closed book, paper-based exam.
  ▪ You may bring one hand-written page of notes that you created (front & back).
  ▪ You will have a study guide attached to your exam.
• Midterm information: http://inst.eecs.berkeley.edu/~cs61a/fa13/exams/midterm1.html
Announcements

• Homework 2 due Tuesday at 11:59pm
• Project 1 due Thursday at 11:59pm
  ▪ Extra debugging office hours in Soda 405: Tuesday 6–8, Wednesday 6–7, Thursday 5–7
  ▪ Readers hold these office hours; they are the ones who give you composition scores!
• Optional guerrilla section Monday 6pm–8pm, meeting outside of Soda 310
• Midterm 1 is next Monday 9/23 from 7pm to 9pm in various locations across campus
  ▪ Closed book, paper-based exam.
  ▪ You may bring one hand-written page of notes that you created (front & back).
  ▪ You will have a study guide attached to your exam.
• Midterm information: http://inst.eecs.berkeley.edu/~cs61a/fa13/exams/midterm1.html
• Review session: Saturday 9/21 (details TBD)
Announcements

• Homework 2 due Tuesday at 11:59pm
• Project 1 due Thursday at 11:59pm
  ▪ Extra debugging office hours in Soda 405: Tuesday 6–8, Wednesday 6–7, Thursday 5–7
  ▪ Readers hold these office hours; they are the ones who give you composition scores!
• Optional guerrilla section Monday 6pm–8pm, meeting outside of Soda 310
• Midterm 1 is next Monday 9/23 from 7pm to 9pm in various locations across campus
  ▪ Closed book, paper-based exam.
  ▪ You may bring one hand-written page of notes that you created (front & back).
  ▪ You will have a study guide attached to your exam.
• Midterm information: http://inst.eecs.berkeley.edu/~cs61a/fa13/exams/midterm1.html
• Review session: Saturday 9/21 (details TBD)
• HKN Review session: Sunday 9/22 (details TBD)
Announcements

• Homework 2 due Tuesday at 11:59pm
• Project 1 due Thursday at 11:59pm
  ▪ Extra debugging office hours in Soda 405: Tuesday 6–8, Wednesday 6–7, Thursday 5–7
  ▪ Readers hold these office hours; they are the ones who give you composition scores!
• Optional guerrilla section Monday 6pm–8pm, meeting outside of Soda 310
• Midterm 1 is next Monday 9/23 from 7pm to 9pm in various locations across campus
  ▪ Closed book, paper-based exam.
  ▪ You may bring one hand-written page of notes that you created (front & back).
  ▪ You will have a study guide attached to your exam.
  ▪ Midterm information: http://inst.eecs.berkeley.edu/~cs61a/fa13/exams/midterm1.html
• Review session: Saturday 9/21 (details TBD)
• HKN Review session: Sunday 9/22 (details TBD)
• Review office hours on Monday 9/23 (details TBD)
Recursive Functions
Recursive Functions
Recursive Functions

**Definition:** A function is called *recursive* if the body of that function calls itself, either directly or indirectly.
Recursive Functions

**Definition:** A function is called *recursive* if the body of that function calls itself, either directly or indirectly.

**Implication:** Executing the body of a recursive function may require applying that function again.
Recursive Functions

**Definition:** A function is called *recursive* if the body of that function calls itself, either directly or indirectly.

**Implication:** Executing the body of a recursive function may require applying that function again.
Recursive Functions

**Definition:** A function is called *recursive* if the body of that function calls itself, either directly or indirectly.

**Implication:** Executing the body of a recursive function may require applying that function again.

*Drawing Hands, by M. C. Escher (lithograph, 1948)*
Digit Sums

\[2+0+1+3 = 6\]
Digit Sums

\[ 2 + 0 + 1 + 3 = 6 \]

- If a number \( a \) is divisible by 9, then \( \text{sum_digits}(a) \) is also divisible by 9.
Digit Sums

If a number $a$ is divisible by 9, then $\text{sum_digits}(a)$ is also divisible by 9.

• Useful for typo detection!

\[2+0+1+3 = 6\]
Digit Sums

If a number $a$ is divisible by 9, then $\text{sum_digits}(a)$ is also divisible by 9.

Useful for typo detection!

$2+0+1+3 = 6$
Digit Sums

2+0+1+3 = 6

• If a number $a$ is divisible by 9, then $\text{sum_digits}(a)$ is also divisible by 9.
• Useful for typo detection!
Digit Sums

If a number $a$ is divisible by 9, then $\text{sum_digits}(a)$ is also divisible by 9.

Useful for typo detection!

Credit cards actually use the Luhn algorithm, which we'll implement after digit_sum.
Sum Digits Without a While Statement
Sum Digits Without a While Statement

```python
def split(n):
    """Split positive n into all but its last digit and its last digit."""
    return n // 10, n % 10
```
def split(n):
    """Split positive n into all but its last digit and its last digit."
    return n // 10, n % 10

def sum_digits(n):
    """Return the sum of the digits of positive integer n."
    """
def split(n):
    """Split positive n into all but its last digit and its last digit."""
    return n // 10, n % 10

def sum_digits(n):
    """Return the sum of the digits of positive integer n."""
    if n < 10:
        return n
def split(n):
    """Split positive n into all but its last digit and its last digit.""
    return n // 10, n % 10

def sum_digits(n):
    """Return the sum of the digits of positive integer n.""
    if n < 10:
        return n
    else:
        all_but_last, last = split(n)
def split(n):
    """Split positive n into all but its last digit and its last digit."""
    return n // 10, n % 10

def sum_digits(n):
    """Return the sum of the digits of positive integer n."""
    if n < 10:
        return n
    else:
        all_but_last, last = split(n)
        return sum_digits(all_but_last) + last
The Anatomy of a Recursive Function

def sum_digits(n):
    """Return the sum of the digits of positive integer n."""
    if n < 10:
        return n
    else:
        all_but_last, last = split(n)
        return sum_digits(all_but_last) + last
The Anatomy of a Recursive Function

- The **def statement header** is similar to other functions

```python
def sum_digits(n):
    """Return the sum of the digits of positive integer n."""
    if n < 10:
        return n
    else:
        all_but_last, last = split(n)
        return sum_digits(all_but_last) + last
```
The Anatomy of a Recursive Function

- The **def statement header** is similar to other functions

```python
def sum_digits(n):
    """Return the sum of the digits of positive integer n."""
    if n < 10:
        return n
    else:
        all_but_last, last = split(n)
        return sum_digits(all_but_last) + last
```
The Anatomy of a Recursive Function

• The **def statement header** is similar to other functions
• Conditional statements check for **base cases**

```python
def sum_digits(n):
    """Return the sum of the digits of positive integer n."""
    if n < 10:
        return n
    else:
        all_but_last, last = split(n)
        return sum_digits(all_but_last) + last
```
The Anatomy of a Recursive Function

- The **def statement header** is similar to other functions
- Conditional statements check for **base cases**

```python
def sum_digits(n):
    """Return the sum of the digits of positive integer n."""
    if n < 10:
        return n
    else:
        all_but_last, last = split(n)
        return sum_digits(all_but_last) + last
```
The Anatomy of a Recursive Function

- The **def statement header** is similar to other functions
- Conditional statements check for **base cases**
- Base cases are evaluated **without recursive calls**

```python
def sum_digits(n):
    """Return the sum of the digits of positive integer n."""
    if n < 10:
        return n
    else:
        all_but_last, last = split(n)
        return sum_digits(all_but_last) + last
```
The Anatomy of a Recursive Function

- The **def statement header** is similar to other functions.
- Conditional statements check for **base cases**.
- Base cases are evaluated **without recursive calls**.

```python
def sum_digits(n):
    """Return the sum of the digits of positive integer n.""
    if n < 10:
        return n
    else:
        all_but_last, last = split(n)
        return sum_digits(all_but_last) + last
```
The Anatomy of a Recursive Function

- The `def statement header` is similar to other functions
- Conditional statements check for `base cases`
- Base cases are evaluated `without recursive calls`
- Recursive cases are evaluated `with recursive calls`

```python
def sum_digits(n):
    '''Return the sum of the digits of positive integer n.'''
    if n < 10:
        return n
    else:
        all_but_last, last = split(n)
        return sum_digits(all_but_last) + last
```
The Anatomy of a Recursive Function

• The **def statement header** is similar to other functions
• Conditional statements check for **base cases**
• Base cases are evaluated **without recursive calls**
• Recursive cases are evaluated **with recursive calls**

```python
def sum_digits(n):
    '''Return the sum of the digits of positive integer n.'''
    if n < 10:
        return n
    else:
        all_but_last, last = split(n)
        return sum_digits(all_but_last) + last
```
The Anatomy of a Recursive Function

• The **def statement header** is similar to other functions
• Conditional statements check for **base cases**
• Base cases are evaluated **without recursive calls**
• Recursive cases are evaluated **with recursive calls**

```python
def sum_digits(n):
    """Return the sum of the digits of positive integer n."""
    if n < 10:
        return n
    else:
        all_but_last, last = split(n)
        return sum_digits(all_but_last) + last
```
(Demo)
Recursion in Environment Diagrams
Recursion in Environment Diagrams

```python
1 def fact(n):
2     if n == 0:
3         return 1
4     else:
5         return n * fact(n-1)
6
7 fact(3)
```

Example: [http://goo.gl/XOP9ps](http://goo.gl/XOP9ps)
Recursion in Environment Diagrams

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

fact(3)
```

Example: http://goo.gl/XOP9ps
Recursion in Environment Diagrams

```
1  def fact(n):
2      if n == 0:
3          return 1
4      else:
5          return n * fact(n-1)
6
7  fact(3)
```

Example: [http://goo.gl/XOP9ps](http://goo.gl/XOP9ps)
Recursion in Environment Diagrams

1 def fact(n):
2     if n == 0:
3         return 1
4     else:
5         return n * fact(n-1)
6
7 fact(3)

• The same function fact is called multiple times.

Example: http://goo.gl/XOP9ps
Recursion in Environment Diagrams

```python
1 def fact(n):
2     if n == 0:
3         return 1
4     else:
5         return n * fact(n-1)
6
7 fact(3)
```

*The same function `fact` is called multiple times.*

Example: [http://goo.gl/XOP9ps](http://goo.gl/XOP9ps)
Recursion in Environment Diagrams

```python
1 def fact(n):
2     if n == 0:
3         return 1
4     else:
5         return n * fact(n-1)
6
7 fact(3)
```

- The same function `fact` is called multiple times.
- Different frames keep track of the different arguments in each call.

Example: [http://goo.gl/XOP9ps](http://goo.gl/XOP9ps)
Recursion in Environment Diagrams

1  def fact(n):
2      if n == 0:
3          return 1
4      else:
5          return n * fact(n-1)
6
7  fact(3)

• The same function fact is called multiple times.
• Different frames keep track of the different arguments in each call.
• What \textit{n} evaluates to depends upon which is the current environment.

Example: http://goo.gl/XOP9ps
Recursion in Environment Diagrams

```
1  def fact(n):
2      if n == 0:
3          return 1
4      else:
5          return n * fact(n-1)
6
7  fact(3)
```

- The same function `fact` is called multiple times.
- Different frames keep track of the different arguments in each call.
- What `n` evaluates to depends upon which is the current environment.

Example: [http://goo.gl/XOP9ps](http://goo.gl/XOP9ps)
Recursion in Environment Diagrams

```python
1 def fact(n):
2     if n == 0:
3         return 1
4     else:
5         return n * fact(n-1)
6
7 fact(3)
```

- The same function `fact` is called multiple times.
- Different frames keep track of the different arguments in each call.
- What `n` evaluates to depends upon which is the current environment.
- Each call to `fact` solves a simpler problem than the last: smaller `n`.

Example: [http://goo.gl/XOP9ps](http://goo.gl/XOP9ps)
Iteration vs Recursion

Example: http://goo.gl/Ngh3Lf
Iteration vs Recursion

Iteration is a special case of recursion

Example: [http://goo.gl/Ngh3Lf](http://goo.gl/Ngh3Lf)
Iteration vs Recursion

Iteration is a special case of recursion

\[ 4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24 \]

Example: [http://goo.gl/Ngh3Lf](http://goo.gl/Ngh3Lf)
Iteration vs Recursion

Iteration is a special case of recursion

\[ 4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24 \]

Using iterative control:

Example: [http://goo.gl/Ngh3Lf](http://goo.gl/Ngh3Lf)
Iteration vs Recursion

Iteration is a special case of recursion

\[ 4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24 \]

Using iterative control:

```python
def fact_iter(n):
    total, k = 1, 1
    while k <= n:
        total, k = total * k, k + 1
    return total
```

Example: http://goo.gl/Ngh3Lf
**Iteration vs Recursion**

Iteration is a special case of recursion

\[4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24\]

**Using iterative control:**

```python
def fact_iter(n):
    total, k = 1, 1
    while k <= n:
        total, k = total * k, k + 1
    return total
```

**Using recursion:**

Example: [http://goo.gl/Ngh3Lf](http://goo.gl/Ngh3Lf)
Iteration vs Recursion

Iteration is a special case of recursion

\[ 4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24 \]

Using iterative control:

```python
def fact_iter(n):
    total, k = 1, 1
    while k <= n:
        total, k = total*k, k+1
    return total
```

Using recursion:

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

Example:  
http://goo.gl/Ngh3Lf
Iteration vs Recursion

Iteration is a special case of recursion

\[ 4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24 \]

Using iterative control:

```python
def fact_iter(n):
    total, k = 1, 1
    while k <= n:
        total, k = total*k, k+1
    return total
```

Using recursion:

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

Math:

Example: http://goo.gl/NgH3Lf
Iteration vs Recursion

Iteration is a special case of recursion

\[ 4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24 \]

Using iterative control:

```python
def fact_iter(n):
    total, k = 1, 1
    while k <= n:
        total, k = total * k, k + 1
    return total
```

Using recursion:

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

Math:

\[ n! = \prod_{k=1}^{n} k \]

Example: [http://goo.gl/Ngh3Lf](http://goo.gl/Ngh3Lf)
Iteration vs Recursion

Iteration is a special case of recursion

\[ 4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24 \]

Using iterative control:

```python
def fact_iter(n):
    total, k = 1, 1
    while k <= n:
        total, k = total * k, k + 1
    return total
```

Using recursion:

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

Math:

\[ n! = \prod_{k=1}^{n} k \]

\[ n! = \begin{cases} 
1 & \text{if } n = 0 \\
 n \cdot (n-1)! & \text{otherwise} 
\end{cases} \]

Example: [http://goo.gl/Ngh3Lf](http://goo.gl/Ngh3Lf)
Iteration vs Recursion

Iteration is a special case of recursion

$4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$

Using iterative control:

```python
def fact_iter(n):
    total, k = 1, 1
    while k <= n:
        total, k = total* k, k+1
    return total
```

Using recursion:

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

Math:

$$n! = \prod_{k=1}^{n} k$$

Names:

Example: http://goo.gl/NgLr3f
Iteration vs Recursion

Iteration is a special case of recursion

\[ 4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24 \]

Using iterative control:

```python
def fact_iter(n):
    total, k = 1, 1
    while k <= n:
        total, k = total * k, k + 1
    return total
```

Using recursion:

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

Math:

\[ n! = \prod_{k=1}^{n} k \]

Names:

\( n, \text{total}, k, \text{fact_iter} \)

Example: [http://goo.gl/NgH3Lf](http://goo.gl/NgH3Lf)
Iteration vs Recursion

Iteration is a special case of recursion

\[ 4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24 \]

Using iterative control:

```python
def fact_iter(n):
    total, k = 1, 1
    while k <= n:
        total, k = total*k, k+1
    return total
```

Using recursion:

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

Math:

\[ n! = \prod_{k=1}^{n} k \]

Names: \( n, \text{total}, k, \text{fact}_\text{iter} \quad \text{Using iterative control:} \)

\[ n! = \begin{cases} 
1 & \text{if } n = 0 \\
(n \cdot (n - 1)!) & \text{otherwise}
\end{cases} \]

Names: \( n, \text{fact} \quad \text{Using recursion:} \)

Example: [http://goo.gl/Ngh3Lf](http://goo.gl/Ngh3Lf)
Verifying Recursive Functions
The Recursive Leap of Faith
The Recursive Leap of Faith

Photo by Kevin Lee, Preikestolen, Norway
The Recursive Leap of Faith

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

Photo by Kevin Lee, Preikestolen, Norway
The Recursive Leap of Faith

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

Is `fact` implemented correctly?
The Recursive Leap of Faith

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

Is `fact` implemented correctly?

1. Verify the base case.
The Recursive Leap of Faith

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

Is `fact` implemented correctly?

1. Verify the base case.

2. Treat `fact` as a functional abstraction!
The Recursive Leap of Faith

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

Is `fact` implemented correctly?

1. Verify the base case.

2. Treat `fact` as a functional abstraction!

3. Assume that `fact(n-1)` is correct.
The Recursive Leap of Faith

def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)

Is fact implemented correctly?

1. Verify the base case.

2. Treat fact as a functional abstraction!

3. Assume that fact(n-1) is correct.

4. Verify that fact(n) is correct, assuming that fact(n-1) correct.
Mutual Recursion
The Luhn Algorithm
The Luhn Algorithm

Used to verify credit card numbers
The Luhn Algorithm

Used to verify credit card numbers

The Luhn Algorithm

Used to verify credit card numbers


1. From the rightmost digit, which is the check digit, moving left, double the value of every second digit; if product of this doubling operation is greater than 9 (e.g., 7 * 2 = 14), then sum the digits of the products (e.g., 10: 1 + 0 = 1, 14: 1 + 4 = 5).
The Luhn Algorithm

Used to verify credit card numbers


1. From the rightmost digit, which is the check digit, moving left, double the value of every second digit; if product of this doubling operation is greater than 9 (e.g., 7 * 2 = 14), then sum the digits of the products (e.g., 10: 1 + 0 = 1, 14: 1 + 4 = 5).

2. Take the sum of all the digits.
The Luhn Algorithm

Used to verify credit card numbers


1. From the rightmost digit, which is the check digit, moving left, double the value of every second digit; if product of this doubling operation is greater than 9 (e.g., $7 \times 2 = 14$), then sum the digits of the products (e.g., 10: $1 + 0 = 1$, 14: $1 + 4 = 5$).

2. Take the sum of all the digits.

| 1 | 3 | 8 | 7 | 4 | 3 |
The Luhn Algorithm

Used to verify credit card numbers


1. From the rightmost digit, which is the check digit, moving left, double the value of every second digit; if product of this doubling operation is greater than 9 (e.g., 7 * 2 = 14), then sum the digits of the products (e.g., 10: 1 + 0 = 1, 14: 1 + 4 = 5).

2. Take the sum of all the digits.

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>8</td>
<td>7</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1+6=7</td>
<td>7</td>
<td>8</td>
<td>3</td>
</tr>
</tbody>
</table>
The Luhn Algorithm

Used to verify credit card numbers


1. From the rightmost digit, which is the check digit, moving left, double the value of every second digit; if product of this doubling operation is greater than 9 (e.g., $7 \times 2 = 14$), then sum the digits of the products (e.g., $10:1 + 0 = 1$, $14:1 + 4 = 5$).

2. Take the sum of all the digits.

<table>
<thead>
<tr>
<th>1</th>
<th>3</th>
<th>8</th>
<th>7</th>
<th>4</th>
<th>3</th>
</tr>
</thead>
</table>
| 2 | 3 | 1+6=7 | 7 | 8 | 3 | = 30
The Luhn Algorithm

Used to verify credit card numbers


1. From the rightmost digit, which is the check digit, moving left, double the value of every second digit; if product of this doubling operation is greater than 9 (e.g., 7 * 2 = 14), then sum the digits of the products (e.g., 10: 1 + 0 = 1, 14: 1 + 4 = 5).

2. Take the sum of all the digits.

```
  1  3  8  7  4  3
  2  3  1+6=7  7  8  3  = 30
```

The Luhn sum of a valid credit card number is a multiple of 10.
The Luhn Algorithm

Used to verify credit card numbers


1. From the rightmost digit, which is the check digit, moving left, double the value of every second digit; if product of this doubling operation is greater than 9 (e.g., 7 * 2 = 14), then sum the digits of the products (e.g., 10: 1 + 0 = 1, 14: 1 + 4 = 5).

2. Take the sum of all the digits.

<table>
<thead>
<tr>
<th>1</th>
<th>3</th>
<th>8</th>
<th>7</th>
<th>4</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>1+6=7</td>
<td>7</td>
<td>8</td>
<td>3</td>
</tr>
</tbody>
</table>

= 30

The Luhn sum of a valid credit card number is a multiple of 10. (Demo)
Recursion and Iteration
Converting Recursion to Iteration
Converting Recursion to Iteration

*Can be tricky:* Iteration is a special case of recursion.
Converting Recursion to Iteration

**Can be tricky:** Iteration is a special case of recursion.

**Idea:** Figure out what state must be maintained by the iterative function.
**Converting Recursion to Iteration**

*Can be tricky:* Iteration is a special case of recursion.

*Idea:* Figure out what state must be maintained by the iterative function.

```python
def sum_digits(n):
    """Return the sum of the digits of positive integer n."""

    if n < 10:
        return n

    else:
        all_but_last, last = split(n)
        return sum_digits(all_but_last) + last
```
Converting Recursion to Iteration

Can be tricky: Iteration is a special case of recursion.

Idea: Figure out what state must be maintained by the iterative function.

```python
def sum_digits(n):
    """Return the sum of the digits of positive integer n.""
    if n < 10:
        return n
    else:
        all_but_last, last = split(n)
        return sum_digits(all_but_last) + last
```

What's left to sum
Converting Recursion to Iteration

Can be tricky: Iteration is a special case of recursion.

Idea: Figure out what state must be maintained by the iterative function.

def sum_digits(n):
    """Return the sum of the digits of positive integer n."""
    if n < 10:
        return n
    else:
        all_but_last, last = split(n)
        return sum_digits(all_but_last) + last

What's left to sum

A partial sum
**Converting Recursion to Iteration**

*Can be tricky*: Iteration is a special case of recursion.

*Idea*: Figure out what state must be maintained by the iterative function.

```python
def sum_digits(n):
    """Return the sum of the digits of positive integer n."""
    if n < 10:
        return n
    else:
        all_but_last, last = split(n)
        return sum_digits(all_but_last) + last
```

(Demo)
Converting Iteration to Recursion
Converting Iteration to Recursion

More formulaic: Iteration is a special case of recursion.
Converting Iteration to Recursion

More formulaic: Iteration is a special case of recursion.

Idea: The state of an iteration can be passed as arguments.
Converting Iteration to Recursion

**More formulaic:** Iteration is a special case of recursion.

**Idea:** The *state* of an iteration can be passed as arguments.

```python
def sum_digits_iter(n):
    digit_sum = 0
    while n > 0:
        n, last = split(n)
        digit_sum = digit_sum + last
    return digit_sum
```
Converting Iteration to Recursion

**More formulaic**: Iteration is a special case of recursion.

**Idea**: The *state* of an iteration can be passed as arguments.

```python
def sum_digits_iter(n):
    digit_sum = 0
    while n > 0:
        n, last = split(n)
        digit_sum = digit_sum + last
    return digit_sum
```

```python
def sum_digits_rec(n, digit_sum):
    if n == 0:
        return digit_sum
    else:
        n, last = split(n)
        return sum_digits_rec(n, digit_sum + last)
```
Converting Iteration to Recursion

More formulaic: Iteration is a special case of recursion.

Idea: The state of an iteration can be passed as arguments.

```python
def sum_digits_iter(n):
    digit_sum = 0
    while n > 0:
        n, last = split(n)
        digit_sum = digit_sum + last
    return digit_sum
```

```python
def sum_digits_rec(n, digit_sum):
    if n == 0:
        return digit_sum
    else:
        n, last = split(n)
        return sum_digits_rec(n, digit_sum + last)
```

Updates via assignment become...
Converting Iteration to Recursion

More formulaic: Iteration is a special case of recursion.

Idea: The state of an iteration can be passed as arguments.

```python
def sum_digits_iter(n):
    digit_sum = 0
    while n > 0:
        n, last = split(n)
        digit_sum = digit_sum + last
    return digit_sum

def sum_digits_rec(n, digit_sum):
    if n == 0:
        return digit_sum
    else:
        n, last = split(n)
        return sum_digits_rec(n, digit_sum + last)
```

Updates via assignment become...

...arguments to a recursive call