Efficiency
Measuring Efficiency
Efficiency

A measure of how much resource consumption a computational task takes.

An analysis of computer programs rather than a technique for writing them.

In computer science, we are concerned with **time** and **space efficiency**

The time efficiency of could determine how long a user has to wait for a webpage to load.

The space efficiency of your algorithm could determine how much **memory** running your application takes.
Orders of Growth
Prepend

```python
def prepend(lst, val):
    """Add VAL to the front of LST."""
    lst.insert(0, val)
```

<table>
<thead>
<tr>
<th>List Size</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td></td>
</tr>
</tbody>
</table>
Exponentiation
Exponentiation

**Goal:** one more multiplication lets us double the problem size

```
def exp(b, n):
    if n == 0:
        return 1
    else:
        return b * exp(b, n-1)
```

\[ b^n = \begin{cases} 
1 & \text{if } n = 0 \\
 b \cdot b^{n-1} & \text{otherwise} 
\end{cases} \]

<table>
<thead>
<tr>
<th>n</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td></td>
</tr>
</tbody>
</table>
Exponentiation

**Goal:** one more multiplication lets us double the problem size

```python
def exp_fast(b, n):
    if n == 0:
        return 1
    elif n % 2 == 0:
        return square(exp_fast(b, n // 2))
    else:
        return b * exp_fast(b, n - 1)
```

```python
def square(x):
    return x * x
```

\[
b^n = \begin{cases} 
1 & \text{if } n = 0 \\
(b^\frac{1}{2})^n & \text{if } n \text{ is even} \\
b \cdot b^{n-1} & \text{if } n \text{ is odd}
\end{cases}
\]

<table>
<thead>
<tr>
<th>n</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>~1</td>
</tr>
<tr>
<td>16</td>
<td>~5</td>
</tr>
<tr>
<td>512</td>
<td>~9</td>
</tr>
</tbody>
</table>
| 1024  | ~10        | (Demo)
Exponentiation

**Goal:** one more multiplication lets us double the problem size

```python
def exp(b, n):
    if n == 0:
        return 1
    else:
        return b * exp(b, n-1)

def exp_fast(b, n):
    if n == 0:
        return 1
    elif n % 2 == 0:
        return square(exp_fast(b, n//2))
    else:
        return b * exp_fast(b, n-1)

def square(x):
    return x * x
```

Linear time:
- Doubling the input **doubles** the time
- 1024x the input takes 1024x as much time

Logarithmic time:
- Doubling the input **increases** the time by one step
- 1024x the input increases the time by only 10 steps
Quadratic Time

Functions that process all pairs of values in a sequence of length n take quadratic time

def overlap(a, b):
    count = 0
    for item in a:
        for other in b:
            if item == other:
                count += 1
    return count

overlap([3, 5, 7, 6], [4, 5, 6, 5])

(Demo)
Exponential Time

Tree-recursive functions can take exponential time

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```
# Common Orders of Growth

<table>
<thead>
<tr>
<th>Order of Growth</th>
<th>Example</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Exponential growth.</strong> E.g., recursive <code>fib</code></td>
<td>Incrementing $n$ multiplies time by a constant</td>
<td>$a \cdot b^{n+1} = (a \cdot b^n) \cdot b$</td>
</tr>
<tr>
<td><strong>Quadratic growth.</strong> E.g., <code>overlap</code></td>
<td>Incrementing $n$ increases time by $n$ times a constant</td>
<td>$a \cdot (n + 1)^2 = (a \cdot n^2) + a \cdot (2n + 1)$</td>
</tr>
<tr>
<td><strong>Linear growth.</strong> E.g., slow <code>exp</code></td>
<td>Incrementing $n$ increases time by a constant</td>
<td>$a \cdot (n + 1) = (a \cdot n) + a$</td>
</tr>
<tr>
<td><strong>Logarithmic growth.</strong> E.g., <code>exp_fast</code></td>
<td>Doubling $n$ only increments time by a constant</td>
<td>$a \cdot \ln(2 \cdot n) = (a \cdot \ln n) + a \cdot \ln 2$</td>
</tr>
<tr>
<td><strong>Constant growth.</strong> Increasing $n$ doesn't affect time</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Order of Growth Notation
### Big Theta and Big O Notation for Orders of Growth

<table>
<thead>
<tr>
<th>Growth Type</th>
<th>Example</th>
<th>Big Theta $\Theta(n)$</th>
<th>Big O $O(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Exponential growth</strong></td>
<td>E.g., recursive fib</td>
<td>$\Theta(b^n)$</td>
<td>$O(b^n)$</td>
</tr>
<tr>
<td>Incrementing $n$ multiplies time by a constant</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Quadratic growth</strong></td>
<td>E.g., overlap</td>
<td>$\Theta(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Incrementing $n$ increases time by $n$ times a constant</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Linear growth</strong></td>
<td>E.g., slow exp</td>
<td>$\Theta(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Incrementing $n$ increases time by a constant</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Logarithmic growth</strong></td>
<td>E.g., exp_fast</td>
<td>$\Theta(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>Doubling $n$ only increments time by a constant</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Constant growth</strong></td>
<td>Increasing $n$ doesn't affect time</td>
<td>$\Theta(1)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>
Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

```python
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```

(Demo)
Memoization Revisited

(Demo)
Memoization

**Idea:** Remember the results that have been computed before

```python
def memo(f):
    cache = {}

def memoized(n):
    if n not in cache:
        cache[n] = f(n)
    return cache[n]
return memoized
```

Keys are arguments that map to return values

Same behavior as f, if f is a pure function

(Demo)
Memoized Tree Recursion

- fib(5)
  - fib(3)
    - fib(1)
      - fib(0) 0
      - fib(1) 1
    - fib(2)
      - fib(0) 0
      - fib(1) 1
  - fib(4)
    - fib(3)
      - fib(1)
        - fib(0) 0
        - fib(1) 1
      - fib(2)
        - fib(0) 0
        - fib(1) 1
    - fib(2)
      - fib(0) 0
      - fib(1) 1
Break
Revisiting Past Problems
Space and Environments

Which environment frames do we need to keep during evaluation?

At any moment there is a set of active environments

Values and frames in active environments consume memory

Memory that is used for other values and frames can be recycled

Active environments:

• Environments for any function calls currently being evaluated
• Parent environments of functions named in active environments
Fibonacci Space Consumption

Assume we have reached this step

fib(5)

fib(3)

fib(1)

1

fib(2)

fib(0)

0

fib(1)

1

fib(4)

fib(2)

fib(1)

0

1

fib(0)

fib(1)

1

fib(3)

fib(1)

1

fib(2)

fib(0)

0

1

Assume we have reached this step
Fibonacci Space Consumption

Assume we have reached this step

fib(5)

Has an active environment
Can be reclaimed
Hasn't yet been created

fib(4)

Assume we have reached this step

fib(3)

fib(2)

fib(1)

fib(0)
Generators and Space

(Demo)