• Suppose that \( f \) is a one-parameter function on real numbers.
• \( O(f) \): functions that eventually grow no faster than \( f \):
  - \( g \in O(f) \) means that \( |g(x)| \leq C \cdot |f(x)| \) for all \( x \geq M \),
  - where \( C \) and \( M \) are constants, generally different for each \( g \).
• \( \Omega(f) \): functions that eventually grow at least as fast as \( f \):
  - \( g \in \Omega(f) \) means that \( f \in O(g) \),
  - so that \( |f(x)| \leq C |g(x)| \) for all \( x > M \), and so
  - \( |g(x)| \geq \frac{C}{2} |f(x)| \).
• \( \Theta(f) \): functions that eventually grow as \( g \) grows:
  - \( \Theta(f) = O(f) \cap \Omega(f) \), so that
  - \( g \in \Theta(f) \) means that \( \frac{1}{2} |f(x)| \leq |g(x)| \leq C |f(x)| \) for all sufficiently large \( x \).

The Notation (II)

• So \( O(f) \), \( \Omega(f) \), and \( \Theta(f) \) are sets of functions.
• If \( E_1(x) \) and \( E_2(x) \) are two expressions involving \( x \), we usually abbreviate \( \lambda x \cdot E_1(x) \in O(\lambda x \cdot E_2(x)) \) as just \( E_1(x) \in O(E_2(x)) \). For example, \( n+1 \in O(n^2) \).
• I write \( f \in O(g) \) where others write \( f = O(g) \), because the latter doesn’t make sense.

Illustration

- Here, \( f \in O(g) \) (\( p = 2 \), see blue line), even though \( f(x) > g(x) \).
  Likewise, \( f \in \Omega(g) \) (\( p = 1 \), see red line), and therefore \( f \in \Theta(g) \).

- That is, \( f(x) \) is eventually (for \( x > M = 1 \)) no more than proportional to \( g(x) \) and no less than proportional to \( g(x) \).

Illustration, contd.

- Here, \( f' \in O(g) \) (\( p = 0.5 \)), even though \( g(x) > f'(x) \) everywhere.

Other Uses of the Notation

- You may have seen \( O(\cdot) \) notation in math, where we say things like
  \[ f(x) = f(0) + f'(0)x + \frac{f''(0)}{2}x^2 + O(x^3), \text{ for } 0 \leq x < a. \]

- Adding or multiplying sets of functions produces sets of functions. The expression to the right of \( \in \) above means "the set of all functions \( g \) such that
  \[ g(x) = f(0) + f'(0)x + \frac{f''(0)}{2}x^2 + h(x) \]
  where \( h(x) \in O(x^3) \)."
Example: Linear Search

Consider the following search function:

```python
def near(L, x, delta):
    """True iff X differs from some member of sequence L by no
    more than DELTA."""
    for y in L:
        if abs(x-y) <= delta:
            return True
    return False
```

There's a lot here we don't know:
- How long is sequence $L$?
- Where in $L$ is $x$ (if it is)?
- What kind of numbers are in $L$ and how long do they take to compare?
- How long do `abs` and `subtraction` take?
- How long does it take to create an iterator for $L$ and how long does its `next` operation take?

So what can we meaningfully say about complexity of `near`?

What to Measure?

If we want general answers, we have to introduce some "strategic vagueness."

Instead of looking at times, we can consider number of "operations."
Which?

The total time consists of:
1. Some fixed overhead to start the function and begin the loop.
2. Per-iteration costs: subtraction, `abs`, `next`, `<=
3. Some cost to end the loop.
4. Some cost to return.

So we can collect total operations into one "fixed-cost operation" (items 1, 3, 4), plus $M(L)$ "loop operations" (item 2), where $M(L)$ is the number of items in $L$ up to and including the $y$ that come within $\delta$ of $x$ (or the length of $L$ if no match).

What Does an "Operation" Cost?

But these "operations" are of different kinds and complexities, so what do we really know?

Assuming that each operation represents some range of possible minimum and maximum values (constants), we can say that:

$$\min_{\text{fixed}} \text{cost} + M(L) \times \min_{\text{loop}} \text{cost} \leq C_{\text{tot}}(L) \leq \max_{\text{fixed}} \text{cost} + M(L) \times \max_{\text{loop}} \text{cost}$$

where $C_{\text{tot}}(L)$ is the cost of `near` on list $L$, and $M(L)$ is the number of items `near` must look at.

Best/Worst Cases

We can simplify by not trying to give results for particular inputs, but instead giving summary results for all inputs of the same "size."

Here, "size" depends on the problem: could be magnitude, length (of list), cardinality (of set), etc.

Since we don't consider specific inputs, we have to be less precise.

Typically, the figure of interest is the worst case over all inputs of the same size.

Since $M(L) \leq \text{len}(L)$, $C_{\text{tot}}(L) \leq \text{len}(L) \times \max_{\text{loop}} \text{cost}$.

So if we let $C_{\text{wc}}(N)$ mean "worst-case cost of `near` over all lists of size $N,"$ we can conclude that

$$C_{\text{wc}}(N) \in O(N)$$

Best of the Worst

But in addition, it's also clear that $C_{\text{wc}}(N) \in \Omega(N)$.

So we can say, most concisely, $C_{\text{wc}}(N) \in \Theta(N)$.

Generally, when a worst-case time is not $\Theta(\cdot)$, it indicates either that
- We don't know (haven't proved) what the worst case really is, just put limits on it, or
- Most often happens when we talk about the worst-case for a problem: "what's the worst case for the best possible algorithm?"
- We know what the worst-case time is, but it's not an easy formula, so we settle for approximations that are easier to deal with.

Example: Nested Loop

Last time, we saw the worst-case $C_{\text{ad}}(N)$ of the nested loop

```python
for i, x in enumerate(L):
    for j, y in enumerate(L, i+1): # Starts at i+1
        if x == y: return True
    if x == y: return True
```

is $\Theta(N^2)$ (where $N$ is the length of $L$).
Example: A Tricky Nested Loop

- What can we say about \( C_{\text{is}}(N) \), the worst-case cost of this function (assume \( \text{pred} \) counts as one constant-time operation):

  ```python
def is_unduplicated(L, pred):
    """True iff the first x in L such that pred(x) is not a duplicate. Also true if there is no x with pred(x)."""
    i = 1
    while i < len(L):
      x = L[i]
      if x == L[i-1]:
        return False
      if i == 1:
        return True
      return False
```

- In this case, despite the nested loop, we read each element of \( L \) at most once. So \( C_{\text{is}}(N) \in \Theta(N) \).

Some Useful Properties

In the following, \( K, k_1, k_0, k_1, \) and \( k_2 \) are constants, and \( N \geq 0 \).

- \( \Theta(K_0 N + K_1) = \Theta(N) \)
- \( \Theta(N^k + N^{k-1}) = \Theta(N^k) \)
- \( \Theta(m|f(N)|) + |g(N)| = \Theta(max(|f(N)|, |g(N)|)) \)
- \( max(|f(N)|, |g(N)|) \leq |f(N)| + |g(N)| \leq 2 max(|f(N)|, |g(N)|) \).
- \( \Theta(\log_b N) = \Theta(\log_d N) \)
- \( \Theta(\log_b N) = \Theta(\log_d N) \)
- \( \Theta(\log_b N) = \Theta(\log_d N) \)
- \( \Theta(|f(N)| + |g(N)|) \neq \Theta(max(|f(N)|, |g(N)|)) \)
- \( \Theta(|f(N)| + |g(N)|) \neq \Theta(max(|f(N)|, |g(N)|)) \)
- \( O(N^{k_1}) < O(k_2^N) \), if \( k_2 > 1 \).
- \( \Theta(k_1 \log_b N, k_0 k_2) = (\log \log_2 N, k_1 k_2 \cdot N \text{ for } N > 0}.

Fast Growth

- Here’s a bad way to see if a sequence appears (consecutively) in another sequence:

  ```python
def is_substring(sub, seq):
    """True iff SUB[0], SUB[1], ... appear consecutively in sequence SEQ."""
    if len(sub) == 0 or sub == seq:
      return True
    elif len(sub) > len(seq):
      return False
    while i < len(L):
      if x > L[m]: low = m+1
      if x < L[m]: high = m
      i = 0
      a duplicate. Also true if there is no x with pred(x)."
  ```

- Suppose we count the number of times \( \text{is_substring} \) is called.
- Then time depends only on \( \text{D=len(seq)\cdot len(sub)} \).
- Define \( C_{\text{is}}(D) \) = worst-case time to compute \( \text{is_substring} \).
- Looking at cases: \( D \leq 0 \) and \( D > 0 \):

  \[
  C_{\text{is}}(D) = \begin{cases} 
  1, & \text{if } D \leq 0 \\
  2C_{\text{is}}(D-1) + 1, & \text{otherwise}
  \end{cases}
  \]

Slow Growth

- A perhaps-familiar technique:

  ```python
def binary_search(L, x):
    """Return True iff X occurs in sorted list L."""
    low, high = 0, len(L)
    while low < high:
      m = (low + high) // 2
      if x < L[m]: high = m
      if x > L[m]: low = m+1
      return False
  ```

- The value of high-low is halved on each iteration, starting from \( N \), the length of \( L \), so counting loop iterations in the worst case:

  \[
  C_{\text{bs}}(N) = \begin{cases} 
  0, & \text{if } N \leq 0 \\
  1 + C_{\text{bs}}(N/2), & \text{otherwise}
  \end{cases}
  \]

- So \( C_{\text{bs}}(N) = 1 + C_{\text{bs}}(N/2) = 1 + 1 + C_{\text{bs}}(N/4) = \cdots \in \Theta(\log N) \)

Some Intuition on Meaning of Growth

- How big a problem can you solve in a given time?
- In the following table, left column shows time in microseconds to solve a given problem as a function of problem size \( N \) (assuming perfect scaling and that problem size takes 1sec).
- Entries show the size of problem that can be solved in a second, hour, month (31 days), and century, for various relationships between time required and problem size.
- \( N = \text{problem size} \)

<table>
<thead>
<tr>
<th>Time (\text{microsec})</th>
<th>1 second</th>
<th>1 hour</th>
<th>1 month</th>
<th>1 century</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \lg N )</td>
<td>( 10^{100000} )</td>
<td>( 10^{1000000000} )</td>
<td>( 10^{8.1\times10^{14}} )</td>
<td>( 10^{9.15\times10^{14}} )</td>
</tr>
<tr>
<td>( N )</td>
<td>( 10^{6} )</td>
<td>( 3.6 \cdot 10^{9} )</td>
<td>( 10^{2.7\times10^{12}} )</td>
<td>( 3.2 \cdot 10^{15} )</td>
</tr>
<tr>
<td>( N \lg N )</td>
<td>( 63000 )</td>
<td>( 1.3 \cdot 10^{9} )</td>
<td>( 10^{7.4\times10^{10}} )</td>
<td>( 6.9 \cdot 10^{13} )</td>
</tr>
<tr>
<td>( N^2 )</td>
<td>( 1000 )</td>
<td>( 60000 )</td>
<td>( 1.6 \cdot 10^{6} )</td>
<td>( 5.6 \cdot 10^{7} )</td>
</tr>
<tr>
<td>( N^3 )</td>
<td>( 100 )</td>
<td>( 1500 )</td>
<td>( 14000 )</td>
<td>( 150000 )</td>
</tr>
<tr>
<td>( 2^N )</td>
<td>( 20 )</td>
<td>( 32 )</td>
<td>( 41 )</td>
<td>( 51 )</td>
</tr>
</tbody>
</table>