An algorithm is a clearly specified set of instructions to solve a problem. Given any algorithm, you should have two questions:

1. Does my algorithm work? (is it provably correct?)
2. Does my algorithm work efficiently?
   a. How much time does it take? (what is its time complexity?)
   b. How much space does it need? (what is its space complexity?)

In this course, we're mostly concerned with time complexity. Proving algorithms correct is the subject of a future course (280--algorithms). We only touch on it occasionally in 151.

Why should we care about the time complexity of an algorithm?

1. To know how fast it is. Duh.
2. To be able to compare it to other algorithms.

Ideas:

1. Implement each candidate.
2. Run them.
3. Compare.
4. Know how fast it is.

Why should we care about the time complexity of an algorithm?

- Is there anything wrong with this approach?

- Time the results.
- Run them.
- Implement each candidate.
- Compare.

So, we want a standardized way of determining the time complexity of an algorithm. We need a function that expresses running time as a function of problem size.

Ideas: 2.

1. Find a function that expresses running time as a function of problem size.
2. To be able to compare it to other algorithms.
3. To know how fast it is.

We want a standardized way of determining the time complexity of an algorithm.

Given any algorithm, you should have two questions:

1. Does my algorithm work?
2. How much time does it take?

Vague details:

1. Count up the major operations in pseudocode, ignoring small details.
2. Find an upper bound on running time, i.e., a worst-case bound.
3. Find an upper bound on running time, i.e., a worst-case bound.

Vague details:

1. Choose an efficient algorithm.
2. Implement it.
3. Run it.
4. Time the results.
5. Compare.
6. Know how fast it is.

Why should we care about the time complexity of an algorithm?
Examples

5

ex.

sum = 0
for i=1 to n do
    sum = sum + i

ex.

for i=1 to n do
    for j=1 to n do
        sum = sum + i*j

ex.

for i=1 to n do
    for j=1 to m do
        a[i][j] = i*j

ex.

for i=1 to n do
    for j=1 to i do
        foo.bar()

ex.

while n > 1 do
    n = n/2

A More Complicated Example

algorithm to get max value in an array

input: an array A of integers
goal/output: return the maximum value in A

max = A[0]
for i=1 to |A|-1 do
    if A[i] > max then
        max = A[i]

return max

The running time may change for different input arrays A.

The running time may change for different input arrays A.

Some of the running time functions we've seen are:

Basic Functions

- constant: \( T(n) = c \)
- logarithmic: \( T(n) = c \log_2(n) + c \)
- linear: \( T(n) = c_1n + c_2 \)
- \( n \log n \): \( T(n) = c_1n \log_2(n) + c_2 \)
- quadratic: \( T(n) = c_1n^2 + c_2n + c_3 \)
- cubic: \( T(n) = c_1n^3 + c_2n^2 + c_3n + c_4 \)
- exponential: \( T(n) = c_1 \times 2^n \)
- factorial: \( T(n) = c_1 \times n! \)

Glass half-full or half-empty?

So, in analyzing time complexity, should we be

optimistic?

We'll mostly focus on worst-case analysis with the occasional average case.

pessimistic?

We call this worst-case analysis.

neither?

We can do average-case analysis.

Computers scan this on all three.

Neither

Pessimistic

Optimistic

So, in analyzing time complexity, we should be

Examples
So really, the bottom line is this:
To determine the big-Oh running time of an algorithm, you should:
• count the number of simple operations (but not incredibly precisely)
• find the dominant term as \( n \) (the input size) goes to infinity

To determine the big-Oh running time of an algorithm, you should:
So really, the bottom line is this:

### Basic Functions

<table>
<thead>
<tr>
<th>Name</th>
<th>Big-Oh</th>
<th>Time-to-process</th>
<th>Max-n-per-day</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant</td>
<td>( O(c) )</td>
<td>1ms</td>
<td></td>
</tr>
<tr>
<td>( 2n )</td>
<td>( O(n) )</td>
<td>1s</td>
<td>86,400,000</td>
</tr>
<tr>
<td>( n \log n )</td>
<td>( O(n \log n) )</td>
<td>9.9s</td>
<td>9,295</td>
</tr>
<tr>
<td>( n^2 )</td>
<td>( O(n^2) )</td>
<td>16.67min</td>
<td>3,943,234</td>
</tr>
<tr>
<td>( n^3 )</td>
<td>( O(n^3) )</td>
<td>11.57days</td>
<td>9,295</td>
</tr>
<tr>
<td>( 2^n )</td>
<td>( O(2^n) )</td>
<td>3.395x10^290ys</td>
<td>442</td>
</tr>
<tr>
<td>( n! )</td>
<td>( O(n!) )</td>
<td>?</td>
<td>26</td>
</tr>
</tbody>
</table>

You’ll want to remember this order (from fastest to slowest).

**Table:**

- You should always try to find the best (i.e. smallest) upper bound.
- Note that although \( f(n) \) is an upper bound, it may not be the tightest bound.

<table>
<thead>
<tr>
<th>( n )</th>
<th>( n^2 )</th>
<th>( n^3 )</th>
<th>( n \log n )</th>
<th>( 2^n )</th>
<th>( n! )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n+1 )</td>
<td>15</td>
<td>16.67min</td>
<td>9,295</td>
<td>3.395x10^290ys</td>
<td>?</td>
</tr>
</tbody>
</table>

\( (n) = \log(n) \). What \( c \) and \( n_0 \) fit the definition?

**Def:** \( f(n) = \Theta(g(n)) \) if there are positive constants \( c_1, c_2 \) such that \( f(n) \leq c_1 g(n) \) for all \( n > n_0 \).

**Under the system, 1.000.000.000.000 + 1.000.000.000.000 is equal to 1.000.000.000.000 + 1.000.000.000.000**

**Under the system, 1.000 is somehow better than 1.000**

Obviously, this is not fool-proof.

You’ll want to always remember this order (from fastest to slowest).