"Simplicity is a great virtue but it requires hard work to achieve it and education to appreciate it. And to make matters worse: complexity sells better."

- Edsger Dijkstra

**What Makes a Good Algorithm?**

Suppose you have two possible algorithms that do the same thing; which is better?

What do we mean by better?
- Faster?
- Less space?
- Easier to code?
- Easier to maintain?
- Required for homework?

**FIRST**, Aim for simplicity, ease of understanding, correctness.

**SECOND**, Worry about efficiency only when it is needed.

**How do we measure speed of an algorithm?**

**Basic Step:** one “constant time” operation

**Constant time operation:** its time doesn’t depend on the size or length of anything. Always roughly the same. Time is bounded above by some number

**Basic step:**
- Input/output of a number
- Access value of primitive-type variable, array element, or object field
- Assign to variable, array element, or object field
- Do one arithmetic or logical operation
- Method call (not counting arg evaluation and execution of method body)

**Counting Steps**

```
// Store sum of 1..n in sum
sum = 0;
// inv: sum = sum of 1...(k-1)
for (int k = 1; k <= n; k = k+1)
{
  sum = sum + k;
}  // Store sum of 1..n in sum
```

```plaintext
Statement: # times done
-- sum = 0; 1
-- k = 1; 1
-- k <= n  n+1
-- k = k+1; n
-- sum = sum + k; n
```

**Total steps:** $3n + 3$

All basic steps take time 1. There are $n$ loop iterations. Therefore, takes time proportional to $n$.

**Linear algorithm in $n$**

**Not all operations are basic steps**

```
// Store n copies of 'c' in s
s = "";
// inv: s contains k-1 copies of 'c'
for (int k = 1; k <= n; k = k+1)
{
  s = s + 'c';
}
```

```
Statement: # times done
-- s = ""; 1
-- k = 1; 1
-- k <= n n+1
-- k = k+1; n
-- s = s + 'c'; n
```

**Total steps:** $3n + 3$

Catenation is not a basic step. For each $k$, catenation creates and fills $k$ array elements.
String Catenation

\[
s = s + "c"; \quad \text{is NOT constant time. It takes time proportional to } 1 + \text{length of } s
\]

Not all operations are basic steps

\[
\text{// Store } n \text{ copies of } 'c' \text{ in } s
s = \"\";
\]

\[
\text{// inv: } s \text{ contains } k\text{-1 copies of } 'c'
\]

\[
\text{for (int } k = 1; k <= n; k= k+1)\{
\text{ } s = s + 'c';
\}
\]

\[
\text{Total steps: } n*(n-1)/2 + 2n + 3
\]

Linear versus quadratic

\[
\text{// Store sum of 1..n in sum}
\text{sum= 0;}
\]

\[
\text{// inv: sum = } \text{sum of 1..(k-1)}
\]

\[
\text{for (int } k = 1; k <= n; k= k+1)
\text{ } \text{sum} = \text{sum + n}
\]

Linear algorithm

Quadratic algorithm

Looking at execution speed

\[
\text{Number of operations executed}
\]

\[
\text{2n+2, } n+2, \text{ and are all linear in } n, \text{ proportional to } n
\]

\[
\text{2n + 2 ops}
\]

\[
n + 2 \text{ ops}
\]

\[
n \text{ ops}
\]

\[
\text{Constant time}
\]

"Big O" Notation

\[
\text{Formal definition: } f(n) \text{ is } O(g(n)) \text{ if there exist constants } c > 0 \text{ and } N \geq 0 \text{ such that for all } n \geq N, \quad f(n) \leq c \cdot g(n)
\]

Intuitively, \( f(n) \) is \( O(g(n)) \) means that \( f(n) \) grows like \( g(n) \) or slower

What do we want from a definition of "runtime complexity"?

1. Distinguish among cases for large \( n \), not small \( n \)
2. Distinguish among important cases, like
   - \( n \cdot n \) basic operations
   - \( n \) basic operations
   - \( \log n \) basic operations
   - 5 basic operations
3. Don't distinguish among trivially different cases. \( 5 \) or 50 operations
   - \( n, n^2, \) or \( 4n \) operations

Get out far enough (for \( n \geq N \)) \( (n) \) is at most \( cg(n) \)
**Prove that \((2n^2 + n)\) is \(O(n^2)\)**

**Formal definition:** \(f(n) = O(g(n))\) if there exist constants \(c > 0\) and \(N \geq 0\) such that for all \(n \geq N\), \(f(n) \leq c \cdot g(n)\)

**Example:** Prove that \((2n^2 + n)\) is \(O(n^2)\)

**Methodology:**
- Start with \(f(n)\) and slowly transform into \(c \cdot g(n)\):
  - Use \(=\), \(<=\), and \(<\) steps
  - At appropriate point, can choose \(N\) to help calculation
  - At appropriate point, can choose \(c\) to help calculation

\[
\begin{align*}
\text{Choose } N &= 1 \\
\text{and } c &= 3
\end{align*}
\]

**Formal definition:** \(f(n) = O(g(n))\) if there exist constants \(c > 0\) and \(N \geq 0\) such that for all \(n \geq N\), \(f(n) \leq c \cdot g(n)\)

**Problem-size examples**

- Suppose a computer can execute 1000 operations per second; how large a problem can we solve?

<table>
<thead>
<tr>
<th>operations</th>
<th>1 second</th>
<th>1 minute</th>
<th>1 hour</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n)</td>
<td>1000</td>
<td>60,000</td>
<td>3,600,000</td>
</tr>
<tr>
<td>(n \log n)</td>
<td>140</td>
<td>4893</td>
<td>200,000</td>
</tr>
<tr>
<td>(n^2)</td>
<td>31</td>
<td>244</td>
<td>1897</td>
</tr>
<tr>
<td>(3n^2)</td>
<td>18</td>
<td>144</td>
<td>1096</td>
</tr>
<tr>
<td>(n^3)</td>
<td>10</td>
<td>39</td>
<td>153</td>
</tr>
<tr>
<td>(2^n)</td>
<td>9</td>
<td>15</td>
<td>21</td>
</tr>
</tbody>
</table>

\(2^n\) is \(O(n^2)\)
**Commonly Seen Time Bounds**

<table>
<thead>
<tr>
<th>Time Bound</th>
<th>Complexity</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>O(1)</td>
<td>constant</td>
<td>excellent</td>
</tr>
<tr>
<td>O(log n)</td>
<td>logarithmic</td>
<td>excellent</td>
</tr>
<tr>
<td>O(n)</td>
<td>linear</td>
<td>good</td>
</tr>
<tr>
<td>O(n log n)</td>
<td>n log n</td>
<td>pretty good</td>
</tr>
<tr>
<td>O(n²)</td>
<td>quadratic</td>
<td>maybe OK</td>
</tr>
<tr>
<td>O(2^n)</td>
<td>exponential</td>
<td>too slow</td>
</tr>
</tbody>
</table>

**Big O Poll**

Consider two different data structures that could store your data: an array or a doubly-linked list. In both cases, let $n$ be the size of your data structure (i.e., the number of elements it is currently storing). What is the running time of each of the following operations:

- get(i) using an array
- get(i) using a DLL
- insert(v) using an array
- insert(v) using a DLL

**Java Lists**

- `java.util` defines an interface `List<E>`
- implemented by multiple classes:
  - `ArrayList`
  - `LinkedList`

**Search for v in b[0..]**

- Q: v is in array b
- Store in i the index of the first occurrence of v in b:
  - R: v is not in b[0..i-1] and b[i] = v.

**Methodology:**
1. Define pre and post conditions.
2. Draw the invariant as a combination of pre and post.
3. Develop loop using 4 loopy questions.

**Practice doing this!**

**The Four Loopy Questions**

- Does it start right?
  - Is $\{Q\}$ init $\{P\}$ true?
- Does it continue right?
  - Is $\{P \&\& B\} S \{P\}$ true?
- Does it end right?
  - Is $P \&\& IB \Rightarrow R$ true?
- Will it get to the end?
  - Does it make progress toward termination?
Search for v in b[0..]

Q: v is in array b

Store in i the index of the first occurrence of v in b:

R: v is not in b[0..i-1] and b[i] = v.

pre: b 0 b.length
post: i 0 i b.length
inv: b i ? v in here

Linear algorithm: O(b.length)

Each iteration takes constant time.

Worst case: b.length iterations

Binary search for v in sorted b[0..]

// b is sorted. Store in i a value to truthify R:
// b[0..i] <= v < b[i+1..]

pre: b sorted b.length
post: i 0 i b.length
inv: b i k ? v in here

Logarithmic: O(log(b.length))

Worst case: log(b.length) iterations

This algorithm is better than binary searches that stop when v is found.
1. Gives good info when v not in b.
2. Works when b is empty.
3. Finds first occurrence of v, not arbitrary one.
4. Correctness, including making progress, easily seen using invariant

Dutch National Flag Algorithm

Each iteration takes constant time.

Worst case: log(b.length) iterations
Asymptotically, which algorithm is faster?

Dutch National Flag Algorithm: invariant P1

Dutch National Flag Algorithm: invariant P2

Asymptotically, which algorithm is faster?