“Progress is made by lazy men looking for easier ways to do things.”
- Robert Heinlein

**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)

**Not all operations are basic steps**

**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;
}

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

```
Statement: # times done
sum = 0;    1
k = 1;      1
k <= n      n+1
k = k+1;    n
sum = sum + k;
Total steps: 3n + 3
```

**Linear algorithm in n**

// 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’;
Total:       3
```

Concatenation is not a basic step. For each k, concatenation creates and fills k array elements.
String Concatenation

```
s = s + "c";
```

is NOT constant time. It takes time proportional to 1 + length of `s`.

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';

Total steps: n*(n-1)/2 + 2n + 3
```

```
// 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';
```

Concatenation is not a basic step. For each `k`, concatenation creates and fills `k` array elements.

Linear versus quadratic

```
// 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;

Linear algorithm
```

```
// 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';

Quadratic algorithm
```

In comparing the runtimes of these algorithms, the exact number of basic steps is not important. What’s important is that:

- One is linear in `n`—takes time proportional to `n`.
- One is quadratic in `n`—takes time proportional to `n^2`.

Looking at execution speed

```
Number of operations executed
0 2n+2, n^2, n are all linear in `n`, proportional to `n`
2n + 2 ops
n + 2 ops
n ops
```

```
2n^2, 2n^2, n are all linear in `n`, proportional to `n`
```

```
0 1 2 3 ... size n of the array
```

"Big O" Notation

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

```
Get out far enough (for $n \geq N$) $f(n)$ is at most $c \cdot g(n)$
```

Intuitively, $f(n)$ is $O(g(n))$ means that $f(n)$ grows like $g(n)$ or slower.
Prove that \( n^2 + n \) is \( O(n^2) \)

**Formal definition:** \( f(n) \) is \( O(g(n)) \) 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 \( \leq \) and \( < \) steps
  - At appropriate point, can choose \( N \) to help calculation
  - At appropriate point, can choose \( c \) to help calculation

- Choose \( N = 1 \) and \( c = 3 \)

**Formal definition:** \( f(n) \) is \( O(g(n)) \) 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 \( O(n^2) \)

\[
\begin{align*}
f(n) &= \text{<definition of } f(n)> \\
&= 2n^2 + n \\
&\leq \text{<for } n \geq 1, n \leq n^2> \\
&= \text{<arithmetic>} \\
&= 3n^2 \\
&= \text{<definition of } g(n) = n^2> \\
&= 3^2 g(n)
\end{align*}
\]

Choose \( N = 1 \) and \( c = 3 \)

Prove that \( 100 n + \log n \) is \( O(n) \)

\[
\begin{align*}
f(n) &= \text{<put in what } f(n) \text{ is}> \\
&= 100 n + \log n \\
&\leq \text{<We know } \log n \leq n \text{ for } n \geq 1> \\
&= \text{<arithmetic>} \\
&= 101 n \\
&= \text{<g(n) = n>} \\
&= 101 g(n)
\end{align*}
\]

Choose \( N = 1 \) and \( c = 101 \)

**O(\ldots) Examples**

- Let \( f(n) = 3n^2 + 6n - 7 \)
  - \( f(n) \) is \( O(n^2) \)
  - \( f(n) \) is \( O(n^3) \)
  - \( f(n) \) is \( O(n^4) \)
  - \( \ldots \)
  - \( p(n) = 4n \log n + 34n - 89 \)
  - \( p(n) \) is \( O(n \log n) \)
  - \( p(n) \) is \( O(n^2) \)
  - \( h(n) = 202^3 + 40n \)
  - \( h(n) = O(2^n) \)
  - \( a(n) = 34 \)
  - \( a(n) = O(1) \)

- **Choose the smallest one**

Do NOT say or write \( f(n) = O(g(n)) \)

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

\( f(n) = O(g(n)) \) is simply WRONG. Mathematically, it is a disaster. You see it sometimes, even in textbooks. Don’t read such things.

Here’s an example to show what happens when we use \( = \) this way.

We know that \( n+2 = O(n) \) and \( n+3 = O(n) \). Suppose we use =

\[
\begin{align*}
n+2 &= O(n) \\
n+3 &= O(n)
\end{align*}
\]

But then, by transitivity of equality, we have \( n+2 = n+3 \).
We have proved something that is false. Not good.

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>
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(n^2)</td>
<td>quadratic</td>
<td>maybe OK</td>
</tr>
<tr>
<td>O(n^3)</td>
<td>cubic</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..]

// Store in i the index of the first occurrence of v in array b
// Precondition: v is in b.

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.

Search for v in b[0..]

// Store in i the index of the first occurrence of v in array b
// Precondition: v is in b.

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.

The Four Loopy Questions

- Does it start right?
  - Is \( (Q) \) init \( (P) \) true?
- Does it continue right?
  - Is \( (P \land B) \) \( (P) \) true?
- Does it end right?
  - Is \( P \land \neg B \rightarrow R \) true?
- Will it get to the end?
  - Does it make progress toward termination?
Search for v in b[0..]

// Store i the index of the first occurrence of v in array b
// Precondition: v is in b.

pre: b

post: b

inv: b

Linear algorithm: O(b.length)

Another way to search for v in b[0..]

// Store i to truthify b[0..i] <= v < b[i+1..]
// Precondition: b is sorted.

pre: b

post: b

inv: b

Logarithmic: O(log(b.length))

Another way to search for v in b[0..]

// Store i to truthify b[0..i] <= v < b[i+1..]
// Precondition: b is sorted.

pre: b

post: b

inv: b

Logarithmic: O(log(b.length))

Dutch National Flag Algorithm

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 last occurrence of v, not arbitrary one.
4. Correctness, including making progress, easily seen using invariant

Logarithmic: O(log(b.length))
Dutch National Flag Algorithm

**Dutch national flag.** Swap b[0..n-1] to put the reds first, then the whites, then the blues. That is, given precondition Q, swap values of b[0..n] to truthify postcondition R:

<table>
<thead>
<tr>
<th>Q: b_1</th>
<th>b_2</th>
<th>...</th>
<th>b_n</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R: b_1</td>
<td>b_2</td>
<td>...</td>
<td>b_n</td>
</tr>
<tr>
<td>reds</td>
<td>whites</td>
<td>blues</td>
<td></td>
</tr>
</tbody>
</table>

P1: Invariant P1

<table>
<thead>
<tr>
<th>h</th>
<th>k</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>n</td>
</tr>
</tbody>
</table>

P1: Invariant P2

<table>
<thead>
<tr>
<th>h</th>
<th>k</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>n</td>
</tr>
</tbody>
</table>

P2: Invariant P2

Asymptotically, which algorithm is faster?

**Invariant 1**

<table>
<thead>
<tr>
<th>h</th>
<th>k</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>n</td>
</tr>
</tbody>
</table>

**Invariant 2**

<table>
<thead>
<tr>
<th>h</th>
<th>k</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>n</td>
</tr>
</tbody>
</table>

These two algorithms have the same asymptotic running time (both are $O(n)$).