“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)
// 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.
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';
}

Concatenation is not a basic step. For each k, catenation 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';
}

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

---

<table>
<thead>
<tr>
<th>Statement:</th>
<th># times</th>
<th># steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>s = &quot;&quot;;</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>k = 1;</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>k &lt;= n</td>
<td>n+1</td>
<td>1</td>
</tr>
<tr>
<td>k = k+1;</td>
<td>n</td>
<td>1</td>
</tr>
<tr>
<td>s = s + 'c';</td>
<td>n</td>
<td>k</td>
</tr>
</tbody>
</table>

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

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

$2n+2$, $n+2$, $n$ are all linear in $n$, proportional to $n$

<table>
<thead>
<tr>
<th>size n of the array</th>
<th>Number of operations executed</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$2n+2$</td>
</tr>
<tr>
<td>1</td>
<td>$n+2$</td>
</tr>
<tr>
<td>2</td>
<td>$n$</td>
</tr>
<tr>
<td>3</td>
<td>$2n+2$</td>
</tr>
</tbody>
</table>

Constant time

$2n + 2$ ops

$n + 2$ ops

$n$ ops
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
   - $n^2$ 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
Formal definition: $f(n)$ is $O(g(n))$ if there exist constants $c > 0$ and $N \geq 0$ such that for all $n \geq N$, $f(n) \leq c \cdot g(n)$

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

Get out far enough (for $n \geq N$) $f(n)$ is at most $c \cdot g(n)$.
Prove that \((2n^2 + n)\) is \(O(n^2)\)

**Formal definition:** \(f(n) \text{ is } O(g(n))\) 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
Prove that \((2n^2 + n)\) 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 \(O(n^2)\)

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

Transform \(f(n)\) into \(c \cdot g(n)\):
- Use \(=, \leq, <\) steps
- Choose \(N\) to help calc.
- Choose \(c\) to help calc

Choose \(N = 1\) and \(c = 3\)
Prove that \( 100 \, n + \log n \) is \( O(n) \)

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

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

Choose \( N = 1 \) and \( c = 101 \)
O(…) Examples

Let \( f(n) = 3n^2 + 6n - 7 \)
- \( f(n) \) is \( O(n^2) \)
- \( f(n) \) is \( O(n^3) \)
- \( f(n) \) is \( O(n^4) \)
- …

\( p(n) = 4n \log n + 34n - 89 \)
- \( p(n) \) is \( O(n \log n) \)
- \( p(n) \) is \( O(n^2) \)

\( h(n) = 20 \cdot 2^n + 40n \)
- \( h(n) \) is \( O(2^n) \)

\( a(n) = 34 \)
- \( a(n) \) is \( O(1) \)

Only the *leading* term (the term that grows most rapidly) matters

If it’s \( O(n^2) \), it’s also \( O(n^3) \) etc! However, we always use the smallest one
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 \( n+2 \) is \( O(n) \) and \( n+3 \) is \( O(n) \). 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.
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>Big O Notation</th>
<th>Time 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>
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..] 

/** returns the index of the first occurrence of v in array b 
 * Precondition: b is sorted **/

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!
/** returns the index of the first occurrence of v in array b
 * Precondition: b is sorted **/

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

```java
/** returns the index of the first occurrence of v in array b
 * Precondition: b is sorted, v is in b
 **/  
```

```java
while (b[i] < v) {
    i = i + 1;
}
return i;
```

Each iteration takes constant time.

Worst case: b.length iterations

Linear algorithm: O(b.length)
Another way to search for v in b[0..]

/** returns the index of the first occurrence of v in array b
* Precondition: b is sorted, 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.

Practice doing this!
Another way to search for $v$ in $b[0..]$

```java
/**
 * returns the index of the first occurrence of $v$ in array $b$
 * Precondition: $b$ is sorted, $v$ is in $b$
 */

k = -1;
i = b.length;
while (k < i - 1) {
    int j = (k + i) / 2;
    if (b[j] < v) k = j;
    else i = j;
}
return i;
```

Each iteration takes constant time.
Worst case: $\log(b.length)$ iterations

<table>
<thead>
<tr>
<th>pre</th>
<th>post</th>
<th>inv</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b$</td>
<td>$&lt; v$</td>
<td>$\geq v$</td>
</tr>
<tr>
<td>$k$</td>
<td>$i$</td>
<td>$b.length$</td>
</tr>
<tr>
<td>$0$</td>
<td>$b.length$</td>
<td>$0$</td>
</tr>
</tbody>
</table>

Logarithmic: $O(\log(b.length))$
Another way to search for \( v \) in \( b[0..] \)

/\*\* returns the index of the first occurrence of \( v \) in array \( b \)\n  * Precondition: \( b \) is sorted \*\*/

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

\[
i = -1; \\
k = b.length; \\
\textbf{while} \ (i < k-1) \ \{ \\
    \text{int } j = (k+i)/2; \\
    \text{if } b[j] < v \ ? \ i = j : k = j \\
\}\n\]

Each iteration takes constant time.
Worst case: \( \log(b.length) \) iterations

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:

Dutch National Flag Algorithm

<table>
<thead>
<tr>
<th>0</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q: b</td>
<td>?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>R: b</td>
<td>reds whites blues</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1: b</td>
<td>reds whites blues ?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2: b</td>
<td>reds whites ? blues</td>
</tr>
</tbody>
</table>
Dutch National Flag Algorithm: invariant P1

h = 0; k = h; p = k;
while (p != n) {
    if (b[p] blue) p = p+1;
    else if (b[p] white) {
        swap b[p], b[k];
        p = p+1; k = k+1;
    }
    else { // b[p] red
        swap b[p], b[h];
        swap b[p], b[k];
        p = p+1; h = h+1; k = k+1;
    }
}
Dutch National Flag Algorithm: invariant P2

\[ h = 0; \quad k = h; \quad p = n; \]

\[ \text{while} \ (k \neq p) \{ \]

\[ \quad \text{if} \ (b[k] \text{ white}) \quad k = k+1; \]
\[ \quad \text{else if} \ (b[k] \text{ blue}) \{ \]
\[ \quad \quad p = p-1; \]
\[ \quad \quad \text{swap} \ b[k], b[p]; \]
\[ \quad \} \]
\[ \quad \text{else} \ \{ // \ b[k] \text{ is red} \]
\[ \quad \quad \text{swap} \ b[k], b[h]; \]
\[ \quad \quad h = h+1; \quad k = k+1; \]
\[ \} \quad k = k+1; \quad p = p-1; \]

Q: \[ b \]

R: \[ b \text{ reds} \quad \text{whites} \quad \text{blues} \]

P2: \[ b \text{ reds} \quad \text{whites} \quad ? \quad \text{blues} \]
Asymptotically, which algorithm is faster?

**Invariant 1**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>h</th>
<th>k</th>
<th>p</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>reds</td>
<td>whites</td>
<td>blues</td>
<td>?</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

h = 0; k = h; p = k; while ( p != n ) {
    if (b[p] blue) p = p+1;
    else if (b[p] white) {
        swap b[p], b[k];
        p = p+1; k = k+1;
    }
    else { // b[p] red
        swap b[p], b[h];
        swap b[p], b[k];
        p = p+1; h = h+1; k = k+1;
    }
}

**Invariant 2**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>h</th>
<th>k</th>
<th>p</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>reds</td>
<td>whites</td>
<td>?</td>
<td>blues</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

h = 0; k = h; p = n; while ( k != p ) {
    if (b[k] white) k = k+1;
    else if (b[k] blue) {
        p = p-1;
        swap b[k], b[p];
    }
    else { // b[k] is red
        swap b[k], b[h];
        swap b[p], b[k];
        h = h+1; k = k+1;
    }
}


Asymptotically, which algorithm is faster?

**Invariant 1**

<table>
<thead>
<tr>
<th>0</th>
<th>h</th>
<th>k</th>
<th>p</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>reds</td>
<td>whites</td>
<td>blues</td>
<td>?</td>
<td></td>
</tr>
</tbody>
</table>

might use 2 swaps per iteration

```
if (b[p] blue) p= p+1;
else if (b[p] white) {
    swap b[p], b[k];
    p= p+1; k= k+1;
}
```

**Invariant 2**

<table>
<thead>
<tr>
<th>0</th>
<th>h</th>
<th>k</th>
<th>p</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>reds</td>
<td>whites</td>
<td>?</td>
<td>blues</td>
<td></td>
</tr>
</tbody>
</table>

uses at most 1 swap per iteration

```
if (b[k] white) k= k+1;
else if (b[k] blue) {
    p= p-1;
}
```

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

```
swap b[p], b[n];
swap b[p], b[k];
p= p+1; h= h+1; k= k+1;
```

```