“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';
}
Total steps: 3n + 3

Not all operations are basic steps. For each k, concatenation creates and fills k array elements.

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

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 \(1 + \text{length of } s\).
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';
}

Statement:                    # times # steps
s = "";                      1  1
k = 1;                      1  1
k <= n                      n+1 1
k = k+1;                    n  1
s = s + 'c';                n  k
Total steps:                n*(n-1)/2 + 2n + 3

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

Quadratic algorithm in n
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

2n + 2 ops
n + 2 ops
n ops

Constant time

size n of the array
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
"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 \((2n^2 + n)\) 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 \(O(n^2)\)

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 \(O(n^2)\)

\[
\begin{align*}
f(n) &= \quad \text{<definition of } f(n) > \\
2n^2 + n &= \quad \text{<arith>}
\end{align*}
\]

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

\[
\begin{align*}
= &= \quad \text{<arith>}
3n^2 &= \quad \text{<definition of } g(n) = n^2 >
\end{align*}
\]

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 $n \geq N$, $f(n) \leq c \cdot g(n)$

$$f(n) = \text{<put in what } f(n) \text{ is}>$$

$$100 \, n + \log n$$

$$\leq \text{<We know } \log n \leq n \text{ for } n \geq 1>$$

$$100 \, n + n$$

$$= \text{<arith> \hspace{1cm} 101 \, n \hspace{1cm} \text{Choose } N = 1 \text{ and } c = 101}$$

$$= \text{<g(n) = n> \hspace{1cm} 101 \, g(n)}$$
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
**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 $n+2$ is $O(n)$ and $n+3$ is $O(n)$. Suppose we use $=$

$n+2 = O(n)$
$n+3 = O(n)$

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>$O(\text{function})$</th>
<th>Complexity</th>
<th>Rating</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>
Java Lists

- `java.util` defines an interface `List<E>`
- implemented by multiple classes:
  - `ArrayList`
  - `LinkedList`
Linear search for $v$ in $b[0..]$ 

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.

```
// Store value in i to truthify b[0..i-1] < v <= b[i..]
// Precondition: b is sorted
```

If $v$ in $b$, set $i$ to index of first occurrence of $v$

If $v$ not in $b$, set $i$ so that all elements of $b$ that are < $v$ are to the left of index $i$.

Practice doing this!
Linear search for v in b[0..]

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

```
// Store value in i to truthify $b[0..i-1] < v \leq b[i..]$
// Precondition: $b$ is sorted
```

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

```
while ( i < b.length &&
        b[i] < v ) { 
    i= i+1;
}  
```

Each iteration takes constant time.

Worst case: $b$.length iterations

Linear algorithm: $O(b$.length$)$
## Binary search for v in b[0..]

```plaintext
// Store value in i to truthify b[0..i-1] < v <= b[i..]
// 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!

<table>
<thead>
<tr>
<th>pre:</th>
<th>post:</th>
<th>inv:</th>
</tr>
</thead>
<tbody>
<tr>
<td>b [0..]</td>
<td>b [0..]</td>
<td>b [0..]</td>
</tr>
<tr>
<td>b sorted</td>
<td>b (&lt; v)</td>
<td>b (&lt; v)</td>
</tr>
<tr>
<td>0 [0..]</td>
<td>i [0..]</td>
<td>k [0..] i</td>
</tr>
<tr>
<td>b.length</td>
<td>b.length</td>
<td>b.length</td>
</tr>
<tr>
<td></td>
<td>(\geq v)</td>
<td>(\geq v)</td>
</tr>
<tr>
<td></td>
<td>(\geq v)</td>
<td>(\geq v)</td>
</tr>
</tbody>
</table>
Binary search for \( v \) in \( b[0..] \)

// Store value in \( i \) to truthify \( b[0..i-1] < v \leq b[i..] \)
// Precondition: \( b \) is sorted

\[
\begin{array}{c|c|c|c}
0 & \text{b.length} & k= -1; \\
\text{pre: } b & \text{sorted} & i= \text{b.length}; \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c}
0 & k & \text{sorted} & \geq v & \text{b.length} \\
\text{inv: } b & \text{sorted} & \geq v \\
\end{array}
\]

Make invariant true initially
Binary search for v in b[0..]

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

post: b[0..i-1] < v <= b[i..]

k = -1;
i = b.length;
while (k < i-1) {
    \}

Determine loop condition B:
!B && inv imply post
Binary search for v in b[0..]

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

inv: b[0..k] < v  sorted  b[i..b.length] ≥ v

k= -1;
i= b.length;
while (k < i-1) {
    int j= (k+i)/2;
    // k < j < i
    Set one of k, i to j
}

Figure out how to make progress toward termination.
Envision cutting size of b[k+1..i-1] in half
Binary search for $v$ in $b[0..]$ 

// Store value in $i$ to truthify $b[0..i-1] < v \leq b[i..]$ 
// Precondition: $b$ is sorted

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

Figure out how to make progress toward termination. 
Cut size of $b[k+1..i-1]$ in half
Binary search for v in b[0..]

// Store value in i to truthify b[0..i-1] < v <= b[i..]
// 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

Logarithmic: \(O(\log(b.\text{length}))\)

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

Each iteration takes constant time.
Worst case: \(\log(b.\text{length})\) iterations
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</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>n</td>
</tr>
</tbody>
</table>

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

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

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

\[
\begin{array}{cccccc}
0 & & & & n \\
\text{Q: } b & & ? \\
0 & & & & n \\
\text{R: } b & \text{reds} & \text{whites} & \text{blues} \\
0 & h & k & p & n \\
\text{P1: } b & \text{reds} & \text{whites} & \text{blues} & ?
\end{array}
\]

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

Q: $b$

R: $b$

P2: $b$

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

while ( $k \neq p$ ) {
  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 $b[k]$, $b[h]$;
    $h = h+1$; $k = k+1$;
  }
}
Asymptotically, which algorithm is faster?

**Invariant 1**

```
0   h   k   p   n
reds whites blues ?
```

```
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**

```
0   h   k   p   n
reds whites ? blues
```

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