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

- Robert Heinlein
• A3 due Friday
• Prelim next Thursday
  • Prelim conflicts: fill out CMS by Friday
  • Review section on Sunday
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;
}

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

Statement:
sum = 0;
k = 1;
k <= n
k = k + 1;
sum = sum + k;

# times done
1
1
n + 1
n
n
3n + 3

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';
}

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

\[ s = s + \text{"c"}; \] is NOT constant time.
It takes time proportional to 1 + length of \( s \)
Not all operations are basic steps

```java
// 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';
}
```

<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: \(n^2/2 + 2n + 3\)

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

Quadratic algorithm in n
Linear versus quadratic

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

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

Number of operations executed vs size n of the array

- Constant time
- $2n + 2$ ops
- $n + 2$ ops
- $n$ ops
- $n^2$ 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.
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
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 \(O(n^2)\)

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

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 $100n + \log n$ 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 $n \geq N$, $f(n) \leq c \cdot g(n)$

$$f(n) = \begin{cases} \text{put in what } f(n) \text{ is} \end{cases}$$

$$100n + \log n \leq \begin{cases} \text{We know } \log n \leq n \text{ for } n \geq 1 \end{cases}$$

$$100n + n = \begin{cases} \text{arith} \end{cases}$$

$$101n$$

$$= \begin{cases} g(n) = n \end{cases}$$

$$101g(n)$$

Choose $N = 1$ and $c = 101$
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** \( f(n) = \mathcal{O}(g(n)) \)

---

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

\( f(n) = \mathcal{O}(g(n)) \) 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 \( \mathcal{O}(n) \) and \( n+3 \) is \( \mathcal{O}(n) \). Suppose we use =

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

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>Big O</th>
<th>Description</th>
<th>Comment</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..]

// 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.

Practice doing this!
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.

Practice doing this!
The Four Loopy Questions

- Does it start right?
  Is \{Q\} \text{ init } \{P\} \text{ true?}

- Does it continue right?
  Is \{P \&\& B\} \text{ S } \{P\} \text{ true?}

- Does it end right?
  Is \text{ P } \&\& \text{ !B } \implies R \text{ true?}

- Will it get to the end?
  Does it make progress toward termination?
Search for \( v \) in \( b[0..] \)

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

```plaintext
i = 0;
while (b[i] != v) {
    i = i + 1;
}
```

Each iteration takes constant time.

Worst case: \( b.length-1 \) iterations

**Linear algorithm: \( O(b.length) \)**
Search for $v$ in sorted $b[0..]$ 

// Store in $i$ to truthify $b[0..i] \leq v < b[i+1..]$ 
// 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!
Another way to search for \( v \) in \( b[0..] \)

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

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

\[
\begin{array}{c|c|c|c}
\text{post: } b & \leq v & > v & k= \text{b.length;}
\end{array}
\]

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

\[
\begin{array}{c|c|c|c|c|c}
\text{b} & \leq v & > v & \text{j = (i+k)/2} \\
0 & i & j & k & \text{b.length} \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c}
\text{while } ( i < k-1 ) \{ \\
\text{int } j= (i+k)/2; \\
\text{if (b[j] \leq v) i= j; } \\
\text{else } k= j; \\
\}
\end{array}
\]
Another way to search for \( v \) in \( b[0..] \)

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

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

Each iteration takes constant time.

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

**Worst case: \( \log(b\text{.length}) \) iterations**
Another way to search for \( v \) in \( b[0..] \)

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

i = 0;
k = b.length;
while (i < k-1) {
    int j = (i+k)/2;
    // i < j < k
    if (b[j] \leq v) i = j;
    else k = j;
}
```

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

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

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

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>?</td>
<td>n</td>
</tr>
</tbody>
</table>

Q: b

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

R: b

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>reds</td>
<td>whites</td>
<td>blues</td>
</tr>
</tbody>
</table>

P1: b

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>h</td>
<td>k</td>
<td>p</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;
    }
}

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

0
Q: b
? n

0
R: b
reds whites blues n

0 h k p n
P2: b
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];
        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>0</td>
<td>h</td>
<td>k</td>
<td>p</td>
<td>n</td>
</tr>
<tr>
<td>reds</td>
<td>whites</td>
<td>blues</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>0</th>
<th>h</th>
<th>k</th>
<th>p</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>h</td>
<td>k</td>
<td>p</td>
<td>n</td>
</tr>
<tr>
<td>reds</td>
<td>whites</td>
<td>?</td>
<td>blues</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];
        h = h+1; k = k+1;
    }
}

}
Asymptotically, which algorithm is faster?

**Invariant 1**

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

**Invariant 2**

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

might use 2 swaps per iteration

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

uses at most 1 swap per iteration

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

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

h= h+1; k= k+1;
```

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