"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)

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.

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;
}
Total steps: 3n + 3

String Concatenation

s= s + "c"; is NOT constant time. It takes time proportional to 1 + length of s

0 \text{ "d"} 1 \text{ "x"} 2 \text{ "c"}
Not all operations are basic steps

```c
// 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

```c
// 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

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 n²

Looking at execution speed

```
Number of operations executed

<table>
<thead>
<tr>
<th></th>
<th>2n+2, n+2, n are all linear in n, proportional to n</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant time</td>
<td>n*n ops</td>
</tr>
<tr>
<td>0</td>
<td>n+2 ops</td>
</tr>
<tr>
<td>1</td>
<td>2n+2 ops</td>
</tr>
<tr>
<td>1</td>
<td>n+2 ops</td>
</tr>
<tr>
<td>1</td>
<td>n*n ops</td>
</tr>
</tbody>
</table>
```

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*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

"Big O" Notation

```
Formal definition: f(n) is O(g(n)) if there exist constants c > 0 and N ≥ 0 such that for all n ≥ N, f(n) ≤ c·g(n)
```

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 ≥ 0 such that for all n ≥ N, f(n) ≤ c·g(n)
```

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

Methodology:
- Start with f(n) and slowly transform into c·g(n):
  - Use = and <= 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 \(O(n^2)\)

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

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

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

**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**

- \(O(1)\) constant excellent
- \(O(\log n)\) logarithmic excellent
- \(O(n)\) linear good
- \(O(n \log n)\) \(n \log n\) pretty good
- \(O(n^2)\) quadratic maybe OK
- \(O(n^3)\) cubic maybe OK
- \(O(2^n)\) exponential too slow
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..]

** 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.

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.

The Four Loopy Questions

Does it start right?
Is \( \{ Q \} \) init \( \{ P \} \) true?

Does it continue right?
Is \( \{ P \land B \} \) S \( \{ P \} \) 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?

Each iteration takes constant time.

Linear algorithm: \( O(b.length) \)

Worst case: \( b.length \) iterations
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 **

<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>≥ v</td>
</tr>
<tr>
<td>sorted</td>
<td>&lt; v</td>
<td>≥ v</td>
</tr>
<tr>
<td>k</td>
<td>i</td>
<td>b.length</td>
</tr>
<tr>
<td>k</td>
<td>i</td>
<td>b.length</td>
</tr>
</tbody>
</table>

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!

Logarithmic: O(log(b.length))

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.length))

Dutch National Flag Algorithm

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

```
while (k < i) {
    if (b[j] < v) i = j;
    else j = (k + j) / 2;
}
```

Each iteration takes constant time.
Worst case: log(b.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:

```
Q: b
R: b
P1: b
P2: b

Dutch National Flag Algorithm: invariant P1
```

0 0
n n
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**

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

R: b

<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>
</tbody>
</table>

P2: b

<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>
</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];
  }
}

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>
</tbody>
</table>

might use 2 swaps per iteration

worst case uses at most 1 swap per iteration

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

**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>
</tbody>
</table>

Asymptotically, which algorithm is faster?