"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

**Prelim Thursday evening**

Sorry about the Sunday review session mixup.

This week’s recitation: review for prelim. Slides are posted on the pinned Piazza note Recitations/Homeworks.

You now know what time time you will take it.

We will announce rooms later, on Thursday.

It has been a nightmare for our admin, Jenna.

Bring your Cornell ID card.

We will scan them as you enter the room.

Those taking course for AUDIT don’t take the prelim

**What Makes a Good Algorithm?**

Suppose you have two possible algorithms that do the same thing; which is *better*?

- Faster?
- Less space?
- Easier to code?
- Easier to maintain?
- Required for homework?

**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;
}
```

**Statement:**

- sum= 0;
- k= 1;
- k <= n
- k= k+1;
- 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:**

- s= "";
- k= 1;
- k <= n
- k= k+1;
- s= s + ‘c’;

**Total steps:** $3n + 3$

---

**Not all operations are basic steps**

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

$s = s + \text{"c"}$; is NOT constant time. It takes time proportional to 1 + length of $s$.

Not all operations are basic steps

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

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

"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 $n \geq N$, $f(n) \leq c \cdot g(n)$
Prove that \((2n^2 + n)\) is \(O(n^2)\)

\[
\begin{align*}
\text{Formal definition: } & f(n) = O(g(n)) \text{ if there exist constants } c > 0 \\
& \text{ and } N \geq 0 \text{ such that for all } n \geq N, \ f(n) \leq c \cdot g(n) \\
\end{align*}
\]

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

Methodology:
- Start with \(f(n)\) and slowly transform into \(c \cdot g(n)\):
  - Use \(=\), \(\leq\), and \(<\) steps
  - At appropriate point, can choose \(N\) to help calculation
  - At appropriate point, can choose \(c\) to help calculation

\[
\begin{align*}
\text{Formal definition: } & f(n) = O(g(n)) \text{ if there exist constants } c > 0 \\
& \text{ and } N \geq 0 \text{ such that for all } n \geq N, \ f(n) \leq c \cdot g(n) \\
\end{align*}
\]

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

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

O\(\ldots\) Examples

Let \(f(n) = 3n^2 + 6n - 7\)

- \(f(n) = O(n^2)\)
- \(f(n) = O(n^3)\)
- \(f(n) = O(n^4)\)
- \(\ldots\)

\(p(n) = 4 n \log n + 34 n - 89\)

- \(p(n) = O(n \log n)\)
- \(p(n) = O(n)\)
- \(h(n) = 20 \cdot 2^n + 40n\)
- \(h(n) = O(2^n)\)
- \(a(n) = 34\)
- \(a(n) = O(1)\)

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

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>

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

\[
\begin{align*}
f(n) & = O(g(n)) \text{ is simply WRONG. Mathematically, it is a disaster.} \\
& \text{You see it sometimes, even in textbooks. Don’t read such things.} \\
\end{align*}
\]

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 = \(n+2 = O(n)\) \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.
### Commonly Seen Time Bounds

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Description</th>
<th>Grade</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(2^n)</td>
<td>exponential</td>
<td>too slow</td>
</tr>
</tbody>
</table>

### Search for v in b[0..]

**Q:** v is in array b  
Store in i the index of the first occurrence of v in b: 
**R:** v is not in b[0..i-1] and b[i] = v.

#### 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!**

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

#### The Four Loopy Questions

- **Does it start right?** Is \((Q) \text{ init } (P) \) true?
- **Does it continue right?** Is \((P \land \& \& B) \Rightarrow (P) \) true?
- **Does it end right?** Is \(P \land \& \& \text{IB} = \Rightarrow R \) true?
- **Will it get to the end?** Does it make progress toward termination?

### Search for v in sorted b[0..]

**Q:** v is in array b  
Store in i the index of the first occurrence of v in b: 
**R:** b[0..i] \leq v < b[i+1].

#### 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!**
11/14/18

Binary search for v in sorted b[0..]

// b is sorted. Store in i a value to truthify R:
//       b[0..i] <= v < b[i+1..]

pre: b
post: b
inv: b

<table>
<thead>
<tr>
<th>pre: b</th>
<th>post: b</th>
<th>inv: b</th>
</tr>
</thead>
<tbody>
<tr>
<td>sorted</td>
<td>≤ v</td>
<td>≤ v</td>
</tr>
<tr>
<td>i</td>
<td>&gt; v</td>
<td>&gt; v</td>
</tr>
<tr>
<td>k</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

// b is sorted. Store in i a value to truthify R:
//       b[0..i] <= v < b[i+1..]

while (i + 1 < k)

i = -1;

k = b.length;

while (i + 1 < k)

int e = (i + k) / 2;
    // -1 ≤ e ≤ k ≤ b.length
    if (b[e] <= v) i = e;
    else k = e;

// -1 ≤ i ≤ e < k
if (b[e] <= v) i = e;
else k = e;

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))
Worst case: log(b.length) iterations

Each iteration takes constant time.

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-1] to truthify postcondition R:

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

Suppose we use invariant P1.
What does the repetend do?
2 swaps to get a red in place

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-1] to truthify postcondition R:

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

Suppose we use invariant P2.
What does the repetend do?
At most one swap per iteration
Compare algorithms without writing code!
Asymptotically, which algorithm is faster?

Invariant 1

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

Invariant 2

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

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