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

- Robert Heinlein

**ASYMPTOTIC COMPLEXITY**

Lecture 10

CS2110 – Fall 2017

**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?

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 argument 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.

### 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 + "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';
}
```

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

Linear versus quadratic

```java
// 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;
```

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

\( n^2 \) ops
\( 2n + 2 \) 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
```

"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 \( n \geq N \), \( f(n) \leq c \cdot g(n) \).
```

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 equal \( \Leftrightarrow \) and \( \leq \) 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 \(O(n^2)\)

<table>
<thead>
<tr>
<th>Formal definition: (f(n) = O(g(n))) if there exist constants (c &gt; 0) and (N \geq 0) such that for all (n \geq N), (f(n) \leq c \cdot g(n))</th>
</tr>
</thead>
</table>

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

<table>
<thead>
<tr>
<th>(f(n))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\leq)</td>
</tr>
<tr>
<td>(2n^2 + n)</td>
</tr>
<tr>
<td>(\leq)</td>
</tr>
<tr>
<td>(3n^2)</td>
</tr>
<tr>
<td>(\leq)</td>
</tr>
<tr>
<td>(3 \cdot g(n))</td>
</tr>
</tbody>
</table>

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

---

Prove that \(100n + \log n\) is \(O(n)\)

<table>
<thead>
<tr>
<th>Formal definition: (f(n) = O(g(n))) if there exist constants (c &gt; 0) and (N \geq 0) such that for all (n \geq N), (f(n) \leq c \cdot g(n))</th>
</tr>
</thead>
</table>

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

<table>
<thead>
<tr>
<th>(f(n))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\leq)</td>
</tr>
<tr>
<td>(100 n + \log n)</td>
</tr>
<tr>
<td>(\leq)</td>
</tr>
<tr>
<td>(100 n + n)</td>
</tr>
<tr>
<td>(\leq)</td>
</tr>
<tr>
<td>(101 n)</td>
</tr>
<tr>
<td>(\leq)</td>
</tr>
<tr>
<td>(101 \cdot g(n))</td>
</tr>
</tbody>
</table>

Choose \(N = 1\) and \(c = 101\)

---

\(O(\ldots)\) 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 \(O(n^2)\)
- \(p(n)\) is \(O(n^3)\)

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

- \(h(n)\) is \(O(2^n)\)

\(a(n) = 34\)

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

---

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

<table>
<thead>
<tr>
<th>Formal definition: (f(n) = O(g(n))) if there exist constants (c &gt; 0) and (N \geq 0) such that for all (n \geq N), (f(n) \leq c \cdot g(n))</th>
</tr>
</thead>
</table>

\(f(n)\) is \(O(g(n))\) 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 \(n+2 = O(n)\) and \(n+3 = O(n^2)\). Suppose we use

\[ n+2 = O(n) \]
\[ n+3 = O(n^2) \]

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

| \(O(1)\) | 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 |
**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!

---

**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, b contains 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!

---

**The Four Loopy Questions**

- Does it start right? Is (Q) init (P) true?
- Does it continue right? Is (P && B) S (P) true?
- Does it end right? Is P && B => R true?
- Will it get to the end? Does it make progress toward termination?

---

**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!

---

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

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

Pre: b sorted b.length

post: b < v ≥ v

inv: b < v sorted b.length

\[i = 0;\]
\[\text{while } (b[i] < v) \{\]
\[i = i + 1;\]
\}\n
Each iteration takes constant time.
Worst case: b.length iterations

Linear algorithm: O(b.length)

---

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

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

Pre: b sorted b.length

post: b < v ≥ v

inv: b < v sorted b.length

\[j = 0;\]
\[k = b.length;\]
\[\text{while } (i < k) \{\]
\[j = (k + i) / 2;\]
\[b[j] < v \Rightarrow i = j; \]
\[k = j;\]
\}\n
Each iteration takes constant time.
Worst case: log(b.length) iterations

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

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 trutify postcondition \( R \):

\[
\begin{array}{c|cccc}
  Q: & b & \text{reds} & \text{whites} & \text{blues} \\
  \hline
  \text{P1:} & b & \text{reds} & \text{whites} & ? & \text{blues} \\
  \text{P2:} & b & \text{reds} & ? & \text{whites} & \text{blues} \\
\end{array}
\]

Use inv \text{P1}:
- at most 2 swaps per iteration.
Use inv \text{P2}:
- at most 1 swap per iteration.

Which Algorithm is better?

<table>
<thead>
<tr>
<th>Invariant 1</th>
<th>Invariant 2</th>
</tr>
</thead>
</table>
| \begin{array}{c|cccc}
  Q: & b & \text{reds} & \text{whites} & \text{blues} \\
  \hline
  \text{P1:} & b & \text{reds} & \text{whites} & ? & \text{blues} \\
  \text{P2:} & b & \text{reds} & ? & \text{whites} & \text{blues} \\
\end{array} & \begin{array}{c|cccc}
  Q: & b & \text{reds} & \text{whites} & ? & \text{blues} \\
  \hline
  \text{P1:} & b & \text{reds} & \text{whites} & ? & \text{blues} \\
\end{array} |
| \begin{array}{c}
  h=0; k=h; p=n; \text{while (} k!=p \text{)} \{ \\
  \begin{array}{c}
  \text{if (} b[k] \text{ white) } k=k+1; \\
  \text{else if (} b[p] \text{ blue) } \\
  \begin{array}{c}
  p=p+1; \\
  \text{swap} \ b[k], \ b[p]; \\
  \end{array}
  \} \\
  \text{else} \{ // b[k] is red \\
  \begin{array}{c}
  \text{swap} \ b[k], \ b[h]; \\
  h=h+1; k=k+1; \\
  \end{array}
  \}
  \}
\end{array} & \begin{array}{c}
  h=0; k=h; p=n; \text{while (} k!=p \text{)} \{ \\
  \begin{array}{c}
  \text{if (} b[p] \text{ blue) } p=p+1; \\
  \text{else if (} b[p] \text{ white) } \\
  \begin{array}{c}
  p=p+1; k=k+1; \\
  \text{swap} \ b[h], \ b[p]; \\
  \end{array}
  \} \\
  \text{else} \{ // b[p] is red \\
  \begin{array}{c}
  \text{swap} \ b[p], \ b[h]; \\
  \text{swap} \ b[p], \ b[k]; \\
  p=p+1; b[h]=b[k]; \\
  \end{array}
  \}
  \}
  \}
\end{array} |

Dutch National Flag Algorithm: invariant \text{P1}

\[
\begin{array}{c|cccc}
  Q: & b & \text{reds} & \text{whites} & \text{blues} \\
  \hline
  \text{P1:} & b & \text{reds} & \text{whites} & ? & \text{blues} \\
  \text{P2:} & b & \text{reds} & ? & \text{whites} & \text{blues} \\
\end{array}
\]

\[
\begin{array}{c|cccc}
  Q: & b & \text{reds} & \text{whites} & \text{blues} \\
  \hline
  \text{P1:} & b & \text{reds} & \text{whites} & ? & \text{blues} \\
\end{array}
\]

Use inv \text{P1}:
- at most 2 swaps per iteration.
Use inv \text{P2}:
- at most 1 swap per iteration.