"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

ASYMPTOTIC COMPLEXITY

Lecture 10
CS2110 – Spring 2018

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: # times done
sum= 0; 1
k= 1; 1
k <= n n+1
k= k+1; n
sum= sum + k; n
Total steps: 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 + "c"; is NOT constant time.
It takes time proportional to 1 + length of s
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.

<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>k</td>
</tr>
</tbody>
</table>

Total steps: \( n \cdot (n-1)/2 + 2n + 3 \)

Linear versus quadratic

```c
// Store sum of 1..n in sum
sum = 0;
// inv: sum contains sum of 1..(k-1)
for (int k = 1; k <= n; k = k+1) {
    sum = sum + n;
}
```

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

<table>
<thead>
<tr>
<th>Number of operations executed</th>
<th>2n+2, n+2, n are all linear in n, proportional to n</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5 ops</td>
</tr>
<tr>
<td>1</td>
<td>2n ops</td>
</tr>
<tr>
<td>2</td>
<td>n+2 ops</td>
</tr>
<tr>
<td>3</td>
<td>n*2 ops</td>
</tr>
<tr>
<td>…</td>
<td>Constant time</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
   - \( n^2 \) 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:** \( f(n) = O(g(n)) \) if there exist constants \( c > 0 \) and \( N \geq 0 \) such that for all \( n \geq N \), \( f(n) \leq c \cdot g(n) \)

```
\[
\text{Get out far enough for } n \geq N \\
\text{Intuitively, } f(n) = O(g(n)) \text{ means that } f(n) \text{ grows like } g(n) \text{ or slower}
\]
```

Prove that \( (2n^2 + n) = O(n^2) \)

**Formal definition:** \( f(n) = O(g(n)) \) 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 \( (2n^2 + n) = O(n^2) \)

**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: \(f(n) = O(g(n))\) 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 \(O(n^2)\)

\[
\begin{align*}
f(n) &= \text{definition of } f(n) > \\quad \text{Use } =, \leq, < \text{ steps } \quad \text{Choose } N \text{ to help calc.} \\
2n^2 + n &= \text{<arith>} \quad \text{Choose } c \text{ to help calc} \\
3n^2 &= \text{<definition of } g(n) = n^2> \\
3^n g(n) &= \text{Choose } N = 1 \text{ and } c = 3 \\
\end{align*}
\]

\(O(\ldots)\) Examples

Let \(f(n) = 3n^2 + 6n - 7\)
- \(f(n) = O(n^2)\)
- \(f(n) = O(n^2)\)
- \(f(n) = O(n)\)
- \(f(n) = \ldots\)
- \(p(n) = 4n \log n + 34n = 89\)
- \(p(n) = O(n \log n)\)
- \(p(n) = O(n^2)\)
- \(h(n) = 20 - 2 + 40n\)
- \(h(n) = O(2^n)\)
- \(o(n) = 34\)
- \(o(n) = O(1)\)

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

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

\(f(n) = O(g(n))\) 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 \(O(n^2)\). Suppose we use

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

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>(O(\ldots))</th>
<th>constant</th>
<th>excellent</th>
<th>linear</th>
<th>good</th>
<th>quadratic</th>
<th>maybe OK</th>
<th>cubic</th>
<th>maybe OK</th>
<th>exponential</th>
<th>too slow</th>
</tr>
</thead>
<tbody>
<tr>
<td>(O(1))</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(O(\log n))</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(O(n))</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(O(n \log n))</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(O(n^2))</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(O(n^3))</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(O(2^n))</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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

- \texttt{java.util} defines an interface \texttt{List\<E\>}
- implemented by multiple classes:
  - \texttt{ArrayList}
  - \texttt{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.

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 \( P \&\& \neg B \Rightarrow R \) 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.

Linear algorithm: \( \Theta(b.\text{length}) \)

Each iteration takes constant time.
Worst case: \( b.\text{length} \) 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

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!

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

Each iteration takes constant time.

\[\text{Worst case: } \log(b.length) \]

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

** 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 \( R \):

\[
\begin{align*}
Q: b & \quad ? & \quad n \\
0 & \quad 0 & \quad n \\
R: b & \quad \text{reds} & \quad \text{whites} & \quad \text{blues} \\
0 & \quad 0 & \quad n \\
P1: b & \quad \text{reds} & \quad \text{whites} & \quad \text{blues} & \quad ? \\
0 & \quad 0 & \quad n \\
P2: b & \quad \text{reds} & \quad \text{whites} & \quad \text{blues} & \quad ? \\
0 & \quad 0 & \quad n
\end{align*}
\]

Dutch National Flag Algorithm: invariant P1

\[
\begin{align*}
0 & \quad \text{h} & \quad k & \quad p & \quad n \\
Q: b & \quad ? & \quad n \\
0 & \quad 0 & \quad n \\
R: b & \quad \text{reds} & \quad \text{whites} & \quad \text{blues} \\
0 & \quad 0 & \quad n \\
P1: b & \quad \text{reds} & \quad \text{whites} & \quad \text{blues} & \quad ? \\
0 & \quad 0 & \quad n \\
P2: b & \quad \text{reds} & \quad \text{whites} & \quad \text{blues} & \quad ? \\
0 & \quad 0 & \quad n
\end{align*}
\]

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 \( R \):

\[
\begin{align*}
Q: b & \quad ? & \quad n \\
0 & \quad 0 & \quad n \\
R: b & \quad \text{reds} & \quad \text{whites} & \quad \text{blues} \\
0 & \quad 0 & \quad n \\
P1: b & \quad \text{reds} & \quad \text{whites} & \quad \text{blues} & \quad ? \\
0 & \quad 0 & \quad n \\
P2: b & \quad \text{reds} & \quad \text{whites} & \quad \text{blues} & \quad ? \\
0 & \quad 0 & \quad n
\end{align*}
\]

Dutch National Flag Algorithm: invariant P1

\[
\begin{align*}
0 & \quad \text{h} & \quad k & \quad p & \quad n \\
Q: b & \quad ? & \quad n \\
0 & \quad 0 & \quad n \\
R: b & \quad \text{reds} & \quad \text{whites} & \quad \text{blues} \\
0 & \quad 0 & \quad n \\
P1: b & \quad \text{reds} & \quad \text{whites} & \quad \text{blues} & \quad ? \\
0 & \quad 0 & \quad n \\
P2: b & \quad \text{reds} & \quad \text{whites} & \quad \text{blues} & \quad ? \\
0 & \quad 0 & \quad n
\end{align*}
\]

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 \( R \):

\[
\begin{align*}
Q: b & \quad ? & \quad n \\
0 & \quad 0 & \quad n \\
R: b & \quad \text{reds} & \quad \text{whites} & \quad \text{blues} \\
0 & \quad 0 & \quad n \\
P1: b & \quad \text{reds} & \quad \text{whites} & \quad \text{blues} & \quad ? \\
0 & \quad 0 & \quad n \\
P2: b & \quad \text{reds} & \quad \text{whites} & \quad \text{blues} & \quad ? \\
0 & \quad 0 & \quad n
\end{align*}
\]
Dutch National Flag Algorithm: invariant P2

```
Q: b
 0 ? n
R: b
 0 h k p n
P2: b
 0 h k p n
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
0 n 0
reds whites blues ?
```

```
Invariant 2
0 n 0
reds whites ? blues
```

```
might use 2 swaps per iteration
uses at most 1 swap per iteration
```

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