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. Therefore, takes time proportional to n.

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, catenation creates and fills k array elements.
String Concatenation

\[ s = s + \text{“}c\text{”}; \]

is NOT constant time.

It takes time proportional to \(1 + \text{length of } s\)

Not all operations are basic steps

// Store n copies of ‘c’ in s
\[
s = \text{“}c\text{”}n;
\]
// inv: s contains k-1 copies of ‘c’
for (int \(k = 1; \ k < \ n; \ k = k + 1\))
    \[ s = s + \text{“}c\text{”}; \]

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

Linear versus quadratic

// Store sum of 1..n in sum
\[
\text{sum} = 0;
\]
// inv: sum = sum of 1..(k-1)
for (int \(k = 1; \ k < \ n; \ k = k + 1\))
    sum = sum + n

Linear algorithm

Looking at execution speed

// Store n copies of ‘c’ in s
\[
s = \text{“}c\text{”}n;
\]
// inv: s contains k-1 copies of ‘c’
for (int \(k = 1; \ k < \ n; \ k = k + 1\))
    \[ s = s + \text{“}c\text{”}; \]

"Big O" Notation

Formal definition: \(f(n) = O(g(n))\) if there exist constants \(c > 0\) and \(N \geq 0\) such that for all \(n \geq N\), \(f(n) \leq c \cdot g(n)\)
**Prove that \((n^2 + n)\) is \(O(n^2)\)**

**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 \(<\) steps
  - At appropriate point, choose \(N\) to help calculation
  - At appropriate point, choose \(c\) to help calculation

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

**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 \(f(n) = O(g(n))\)**

\(f(n) = O(g(n))\) 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 \(n^2 + 2\) is \(O(n)\) and \(n^3 + 3\) is \(O(n)\). Suppose we use \(n^2 + 2 = O(n)\)
\(n^3 + 3 = O(n)\)
But then, by transitivity of equality, we have \(n^2 + 3 = n^3\).
We have proved something that is false. Not good.
<table>
<thead>
<tr>
<th>Time Bound</th>
<th>Complexity</th>
<th>Evaluation</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>

**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 $b[0..]$.**

// 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 $(Q)$ init $(P)$ true?
- Does it continue right?
  - Is $(P \land B) \Rightarrow (P)$ true?
- Does it end right?
  - Is $P \land B \Rightarrow R$ 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 $b$.  

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

Each iteration takes constant time.  

Worst case: $b.length - 1$ iterations  

Linear algorithm: $O(b.length)$  

---  

Another way to search for $v$ in $b[0..]$  

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

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

Each iteration takes constant time.  

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

Logarithmic: $O(\log(b.length))$  

---  

Another way to search for $v$ in $b[0..]$  

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

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

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: $O(\log(b.length))$  

---  

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

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

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

Each iteration takes constant time.  

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

Logarithmic: $O(\log(b.length))$  

Dutch National Flag Algorithm  

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!
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 $R$:

$\begin{align*}
Q: & b \\
R: & b \\
P1: & b \\
P2: & b \\
\end{align*}$

Dutch National Flag Algorithm: invariant P1

$\begin{align*}
Q: & b \\
R: & b \\
P1: & b \\
\end{align*}$

Dutch National Flag Algorithm: invariant P2

$\begin{align*}
Q: & b \\
R: & b \\
P2: & b \\
\end{align*}$

Asymptotically, which algorithm is faster?

$\begin{align*}
\text{Invariant 1} \\
\text{Invariant 2} \\
\end{align*}$

Asymptotically, which algorithm is faster?

$\begin{align*}
\text{Invariant 1} \\
\text{Invariant 2} \\
\end{align*}$

Asymptotically, which algorithm is faster?