Quotes about loops

“O! Thou hast damnable iteration and art, indeed, able
to corrupt a saint.” Shakespeare, Henry IV, Pt I, 1 ii

“Use not vain repetition, as the heathen do.”
Matthew V, 48

Your “if” is the only peacemaker; much virtue in “if”.
Shakespeare, As You Like It.

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?

How do we measure time
and space of an algorithm?

Your time is most important!

FIRST, Aim for simplicity,
ease of understanding,
correctness.

SECOND, Worry about
efficiency only when it is
needed.

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)

Basic Step: one “constant time” operation

// 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;
All operations are basic steps,
take constant time.
There are n loop iterations.
Therefore, takes time proportional to n.
Linear algorithm in n

// 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;
All operations are basic steps,
take constant time.
There are n loop iterations.
Therefore, takes time proportional to n.
Linear algorithm in n

Basic Step: one “constant time” operation

// 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;
All operations are basic steps,
take constant time.
There are n loop iterations.
Therefore, takes time proportional to n.
Linear algorithm in n

// Store n copies of 'c' in s
s= "\\c";
// inv: s contains k-1 copies of 'c'
for (int k= 1; k <= n; k= k+1)
  s= s + 'c';
All operations are basic steps,
except for catenation. For each k,
catenation creates and fills k array
elements. Total number created:
\( 1 + 2 + 3 + \cdots + n \), or
\( n(n+1)/2 \) = \( n^2/2 + 1/2 \)
Quadratic algorithm in n

Linear versus quadratic

// Store n copies of 'c' in s
s= "\\c";
// inv: s contains k-1 copies of 'c'
for (int k= 1; k <= n; k= k+1)
  s= s + 'c';
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 \( n^2 \)
Linear search for v in b[0..]

<table>
<thead>
<tr>
<th>pre: b</th>
<th>0</th>
<th>b.length</th>
</tr>
</thead>
<tbody>
<tr>
<td>post: b</td>
<td>v not here</td>
<td>v</td>
</tr>
</tbody>
</table>

Methodology:
1. Draw the invariant as a combination of pre and post.
2. Develop loop using 4 loopy questions.

Practice doing this!

Once you get the knack of doing this, you will know these algorithms not because you memorize code but because you can develop them at will from the pre- and post-conditions.

Linear search as in problem set: b is sorted

<table>
<thead>
<tr>
<th>pre: b</th>
<th>0</th>
<th>b.length</th>
</tr>
</thead>
<tbody>
<tr>
<td>post: b</td>
<td>&lt;= v</td>
<td>?</td>
</tr>
</tbody>
</table>

inv: b[0] > v? one iteration.

Linear search for v in b[0..]

<table>
<thead>
<tr>
<th>pre: b</th>
<th>v in here</th>
</tr>
</thead>
<tbody>
<tr>
<td>post: b</td>
<td>v not here</td>
</tr>
</tbody>
</table>

while (b[h] != v) {
    h = h + 1;
}

Each iteration takes constant time.

In the worst case, requires b.length iterations.

Worst case time: proportional to b.length.

Average (expected) time: A little statistics tells you b.length/2 iterations, still proportional to b.length

b is sorted --- use a binary search?

<table>
<thead>
<tr>
<th>pre: b</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>post: b</td>
<td>&lt;= v</td>
</tr>
</tbody>
</table>

inv: b[0] > v? one iteration.

As in a dictionary search

Binary search for v in b: b is sorted

<table>
<thead>
<tr>
<th>pre: b</th>
<th>0</th>
<th>b.length</th>
</tr>
</thead>
<tbody>
<tr>
<td>post: b</td>
<td>&lt;= v</td>
<td>&gt; v</td>
</tr>
</tbody>
</table>

while (h + 1 < t) {
    if (b[h+1] <= v) h = h + 1;
    else t = h + 1;
}

If (b[e] <= v) h = e;
else t = e;

Each iteration cuts the size of the ? segment in half.

Binary search: an O(log n) algorithm

<table>
<thead>
<tr>
<th>pre: b</th>
<th>0</th>
<th>h</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>post: b</td>
<td>&lt;= v</td>
<td>?</td>
<td>&gt; v</td>
</tr>
</tbody>
</table>

while (h != t - 1) {
    int e = (h + t) / 2;
    if (b[e] <= v) h = e;
    else t = e;
}

n = 2^k? About k iterations

Each iteration cuts the size of the ? segment in half.

A logarithmic algorithm
Binary search for \( v \) in \( b \): \( b \) is sorted

**Pre:**
- \( b \)
- \( b.length \)

**Post:**
- \( b \)
- \( b.length \)
- \( h <= v \)
- \( > v \)

**Inv:**
- \( b \)
- \( h \)
- \( t \)
- \( b.length \)
- \( <= v \)
- \( ? \)
- \( > v \)

```plaintext
h = -1; t = b.length;
while (h != t - 1) {
    int e = (h + t) / 2;
    // h < e < t
    if (b[e] <= v) h = e;
    else t = 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 rightmost occurrence of \( v \) not arbitrary one.
4. Correctness, including making progress, easily seen using invariant

Looking at execution speed

Process an array of size \( n \)

- \( 2n+2, n+2, n \) ops for \( n \) basic operations
- \( n^2 \) ops for \( n \times n \) basic operations
- \( 2n+2, n+2, n \) ops for \( n \) basic operations

What do we want from a definition of “runtime complexity”? (2)

1. Distinguish among cases for large \( n \), not small \( n \)
2. Distinguish among important cases, like
   - \( n \times n \) basic operations
   - \( n \) basic operations
   - \( \log n \) basic operations
   - \( 5 \) basic operations
3. Don’t distinguish among trivially different cases.
   - 5 or 50 operations
   - \( n, n^2, 4n \) operations

Definition of \( O(\ldots) \)

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

Graphical view

Get out far enough (for \( n \geq N \))
\( f(n) \) is at most \( c \cdot g(n) \)

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 \( (n^2 + n) \) is \( O(n^2) \)

**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 \((n^2 + n)\) is \(O(n^2)\)

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

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

Choose \(N = 1\) and \(c = 2\)

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

**Example:**

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

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

But then, by transitivity of equality, we have \(n^2 = n^3\). We have proved something that is false. Not good.

**O(…) Examples**

\[ f(n) = 3n^2 + 6n - 7 \]
- \(f(n)\) is \(O(n^2)\)
- \(f(n)\) is \(O(n^3)\)
- \(f(n)\) is \(O(n^4)\)
- ...\n
\[ p(n) = 4n \log n + 34n - 89 \]
- \(p(n)\) is \(O(n \log n)\)
- \(p(n)\) is \(O(n^2)\)
- \(h(n)\) is \(O(2^n)\)
- \(a(n)\) is \(34\)
- \(a(n)\) is \(O(1)\)

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.

**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 | OK |
| \(O(n^3)\) | cubic | maybe OK |
| \(O(2^n)\) | exponential | too slow |

**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>
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 $R$:

$$Q: b \quad R: b$$

(values in $0..n-1$ unknown)

Four possible simple invariants. We show two.

Dutch National Flag Algorithm: invariant $P_1$

$$Q: b \quad R: b$$

$h=0; k= h; p= k;$

while ($p != n$) {
    if ($b[p]$ blue) $p= p+1;$
    else if ($b[p]$ white) {
        $p= p+1;$
        $k= k+1;$
    } else {
        // $b[p]$ red
        //REQUIRES
        // TWO SWAPS
        swap $b[p], b[k];$
        $p= p+1;$
        $k= k+1;$
    } 
}

$h= 0; k= h; p= n;$

while ($k != p$) {
    if ($b[k]$ white) $k= k+1;$
    else if ($b[p]$ blue) {
        $p= p+1;$
        swap $b[k], b[p];$
    } else {
        // $b[k]$ is red
        swap $b[k], b[h];$
        $h= h+1;$
        $k= k+1;$
    } 
}

Dutch National Flag Algorithm: invariant $P_2$

$$Q: b \quad R: b$$

$h=0; k= h; p= n;$

while ($k != p$) {
    if ($b[k]$ white) $k= k+1;$
    else if ($b[p]$ blue) {
        $p= p+1;$
        swap $b[k], b[p];$
    } else {
        // $b[k]$ is red
        swap $b[k], b[h];$
        $h= h+1;$
        $k= k+1;$
    } 
}

Use inv $P_1$: perhaps 2 swaps per iteration.
Use inv $P_2$: at most 1 swap per iteration.