Help in providing code coverage

White-box testing: make sure each part of program is “exercised” in at least one test case. Called code coverage.

Eclipse has a tool for showing you how good your code coverage is! Use it on A3 (and any programs you write)
JavaHyperText entry:
  code coverage
We demo 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?
- Simpler?
- 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.
// Store n copies of 'c' in s
s = \text{""};
// inv: s contains k-1 copies of 'c'
for (int k = 1; k <= n; k = k+1) {
    s = s + 'c';
}

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

Statement: 
# times done
s = \text{""}; 1
k = 1; 1
k \leq n; n+1
k = k+1; n
s = s + 'c'; n

Total steps: 3n + 3

Basic steps executed in s= s + 'c';

1. Create new String object, say C basic steps.
2. Copy k chars from object s to the new object: k basic steps
3. Place char 'c' into the new object: 1 basic step.
4. Store pointer to new object into s: 1 basic step.

Total of (C+2) + k basic steps.

In the algorithm, s= s + 'c'; is executed n times:
s = s + 'c'; with length of s = 0
s = s + 'c'; with length of s = 1
...
s = s + 'c'; with length of s = n-1

Total of n(C+2) + (0 + 1 + 2 + … n-1) basic steps

Basic steps executed in s = s + 'c';

2

// Store n copies of 'c' in s
s = \text{""};
// inv: s contains k-1 copies of 'c'
for (int k = 1; k <= n; k = k+1) {
    s = s + 'c';
}

Total steps: 2n + 3 + n(C+2) + n^2/2 – n/2 basic steps

for s = s + 'c';

Not all operations are basic steps

// Store n copies of 'c' in s
s = \text{""};
// inv: s contains k-1 copies of 'c'
for (int k = 1; k <= n; k = k+1) {
    s = s + 'c';
}

Total steps: 2n + 3 + n(C+2) + n^2/2 – n/2 basic steps.

For s = s + 'c';

Quadratic algorithm in n

Basic steps executed in s = s + 'c';

// Store n copies of 'c' in s
s = \text{""};
// inv: s contains k-1 copies of 'c'
for (int k = 1; k <= n; k = k+1) {
    s = s + 'c';
}

Total steps: 2n + 3 + n(C+2) + n^2/2 – n/2 basic steps.

for s = s + 'c';

Quadratic algorithm in n

Basic steps executed in s = s + 'c';

String Catenation

s= s + 'c'; is NOT constant time.
It takes time proportional to 1 + length of s

Basic steps executed in s = s + 'c';

s = s + 'c'; // Suppose length of s is k

1. Create new String object, say C basic steps.
2. Copy k chars from object s to the new object: k basic steps
3. Place char 'c' into the new object: 1 basic step.
4. Store pointer to new object into s: 1 basic step.

Total of (C+2) + k basic steps.

In the algorithm, s= s + 'c'; is executed n times:
s = s + 'c'; with length of s = 0
s = s + 'c'; with length of s = 1
...
s = s + 'c'; with length of s = n-1

Total of n(C+2) + (0 + 1 + 2 + … n-1) basic steps

Basic steps executed in s = s + 'c';

// Store n copies of 'c' in s
s = \text{""};
// inv: s contains k-1 copies of 'c'
for (int k = 1; k <= n; k = k+1) {
    s = s + 'c';
}

Total steps: 3n + 3

Not all operations are basic steps
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 Algorithm

- 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;
  ```

### Quadratic Algorithm

- 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';
  ```

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>( n^2 ) ops</td>
<td>( 2n + 2 ) ops</td>
</tr>
<tr>
<td>( n ) ops</td>
<td>( n + 2 ) ops</td>
</tr>
</tbody>
</table>

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 \( n^2 \)

### What do we want from a definition of “runtime complexity”? (Slide 12)

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 4\( n \) operations

### Prove that \( (2n^2 + n) \) 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 = and \( \leq \) and \( < \) steps
  - At appropriate point, can choose \( N \) to help calculation
  - At appropriate point, can choose \( c \) to help calculation

#### Proof:

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

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

### "Big O" Notation

- Get out far enough for \( n \geq N \)
- \( f(n) \) is at most \( c \cdot g(n) \)
  - Intuitively, \( f(n) \) is \( O(g(n)) \) means that \( f(n) \) grows like \( g(n) \) or slower

#### 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 \( O(n^2) \)

#### 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:

- Transform \( f(n) \) into \( c \cdot g(n) \):
  - Use \( =, \leq, < \) steps
  - Choose \( N \) to help calc.
  - Choose \( c \) to help calc

Choose \( N = 1 \) and \( c = 3 \)
Prove that \(100n + \log n\) is \(O(n)\)

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

\[f(n) = \begin{cases} 
\text{<put in what f(n) is>} & \text{100n + log n} \\
\text{<arith>} & \text{Choose N = 1 and c = 101} \\
\text{<g(n) = n>} & \text{101 g(n)} \\
\end{cases}
\]

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

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

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

Commonly Seen Time Bounds

\[
\begin{array}{|l|l|l|}
\hline
\text{O(1)} & \text{constant} & \text{excellent} \\
\text{O(log n)} & \text{logarithmic} & \text{excellent} \\
\text{O(n)} & \text{linear} & \text{good} \\
\text{O(n log n)} & \text{n log n} & \text{pretty good} \\
\text{O(n^2)} & \text{quadratic} & \text{maybe OK} \\
\text{O(n^3)} & \text{cubic} & \text{maybe OK} \\
\text{O(2^n)} & \text{exponential} & \text{too slow} \\
\hline
\end{array}
\]

O(\ldots) Examples

Let \(f(n) = 3n^2 + 6n - 7\)
- \(f(n)\) is \(\Omega(n^2)\)
- \(f(n)\) is \(\Omega(n^3)\)
- \(f(n)\) is \(\Omega(n^4)\)

\(p(n) = 4n \log n + 34n - 89\)
- \(p(n)\) is \(O(n \log n)\)
- \(p(n)\) is \(O(n^2)\)
- \(h(n) = 20 \cdot 2^n + 40n\)
- \(h(n)\) is \(O(2^n)\)
- \(a(n) = 34\)
- \(a(n)\) 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

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>(n^3)</td>
<td>18</td>
<td>144</td>
<td>1096</td>
</tr>
<tr>
<td>(2^n)</td>
<td>9</td>
<td>15</td>
<td>21</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!
**Search for v in b[0..]**

Q: v is in array b

Store i in 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!**

---

**The Four Loopy Questions**

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

---

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

// b is sorted. Store in i a value to trutify R:

// b[0..i] <= v < b[i+1..]

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

Each iteration takes constant time.

Worst case: b.length iterations

---

**Logarithmic: O(log(b.length))**

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

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

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

Q: b

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

R: b

reds whites blues

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

P1: b

reds whites blues

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

P2: b

reds whites blues

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

P1: b

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

P2: b

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

Correctness, including making progress, easily seen using invariant

Dutch National Flag Algorithm: invariant P1

Q: b

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

Q: b

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

R: b

reds whites blues

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

P1: b

reds whites blues ?

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

P2: b

reds whites blues

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

Correctness, including making progress, easily seen using invariant

Dutch National Flag Algorithm: invariant P2

Q: b

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

Q: b

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

R: b

reds whites blues

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

P1: b

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

P2: b

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?

<table>
<thead>
<tr>
<th>Invariant 1</th>
<th>Invariant 2</th>
</tr>
</thead>
</table>
| 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
    swap b[p], b[k];
    p= p+1; h=h+1; k= k+1;
  }
} |
| 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];
    swap b[k], b[p];
    h= h+1; k= k+1;
  }
} |

<table>
<thead>
<tr>
<th>紅</th>
<th>白</th>
<th>青</th>
<th>？</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>h</td>
<td>k</td>
<td>p</td>
</tr>
</tbody>
</table>

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

- Invariant 1 might use 2 swaps per iteration.
- Invariant 2 uses at most 1 swap per iteration.