I have graded Q1, Q2, and A2. Monday morning, they will be placed in the Carpenter basement, to be picked up when a consultant is there.

Quotes for the Day:
Instead of trying out computer programs on test cases until they are debugged, one should prove that they have the desired properties.
John McCarthy, 1961, A basis for a mathematical theory of computation.

Testing may show the presence of errors, but never their absence.
Dijkstra, Second NATO Conf. on Software Engineering, 1969.
On “fixing the invariant”

// {s is the sum of 1..h}

s = s + (h + 1);

h = h + 1;

// {s is the sum of 1..h}
On “fixing the invariant”

// {s is the sum of h..n}  \( s = 5 + 6 + 7 + 8 \)  \( h = 5, \ n = 8 \)
\( s = s + (h-1); \)  
\( h = h-1; \)

// {s is the sum of h..n}  \( s = 4 + 5 + 6 + 7 + 8 \)  \( h = 4, \ n = 8 \)
Loop pattern to process a range m..n–1  
(if m = n, the range is empty)

```c
int h = m;
// invariant: m..h–1 has been processed
while (h != n) {
    Process h;
    h = h + 1;
}
// { m..n–1 has been processed }
```
Loop pattern to process a range \( m..n \)
(if \( m = n+1 \), the range is empty)

```c
int h = m;
// invariant: \( m..h-1 \) has been processed
while (h != n+1) {
    Process h;
    h = h+1;
}
// { m..n has been processed }
```
Loop pattern to process a range m..n in reverse order
(if m = n+1, the range is empty)

```c
int h = n+1;
// invariant:  h..n has been processed (in reverse)
while (h != m) {
    Process h–1;
    h = h–1;
} // { m..n has been processed (in reverse)}
```
Logarithmic algorithm to calculate \( b^{*n} \), for \( c \geq 0 \) (i.e. \( b \) multiplied by itself \( c \) times)

/** set \( z \) to \( b^{*c} \), given \( c \geq 0 \) */

```c
int x = b; int y = c; int z = 1;
// invariant: \( z \times x**y = b^{*c} \) and \( 0 \leq y \leq c \)
while (y != 0) {
    if (y % 2 == 0) {
        x = x * x; y = y/2;
    } else {
        z = z * x; y = y - 1;
    }
}
// { \( z = b^{*c} \) }
```

2**\( n \) in binary is: 1 followed by \( n \) zeros. 2**15 is 32768 (in decimal). 32768 = 2**15 is 15.
Logarithmic algorithm to calculate \( b^{**}c \), for \( c \geq 0 \)
(i.e. \( b \) multiplied by itself \( c \) times)

```c
/** set \( z \) to \( b^{**}c \), given \( c \geq 0 \) */
int x = b; int y = c; int z = 1;
// invariant: \( z * x^{**}y = b^{**}c \) and \( 0 \leq y \leq c \)
while (y != 0) {
    if (y % 2 == 0) {
        x = x * x; y = y/2;
    } else {
        z = z * x; y = y - 1;
    }
}
// { z = b^{**}c }
```

The algorithm looks at the binary representation of \( y \).

- Testing if \( y \) is even means testing whether it rightmost bit is 0.
- \( y = y/2 \); is done by deleting the rightmost bit.
- \( y = y - 1 \); in the algorithm is done by changing the rightmost bit from 1 to 0.
Logarithmic algorithm to calculate $b^{**}c$, for $c \geq 0$
(i.e. $b$ multiplied by itself $c$ times)

/** set $z$ to $b^{**}c$, given $c \geq 0 */

```c
int x = b; int y = c; int z = 1;
// invariant: $z * x^{**}y = b^{**}c$ and $0 \leq y \leq c$
while (y != 0) {
    if (y % 2 == 0) {
        x = x * x; y = y/2;
    }
    else {
        z = z * x; y = y - 1;
    }
}
// { $z = b^{**}c$ }
```

The algorithm is “logarithmic in $c$”
which means that if $c = 2^{**}k$, it takes time proportional to $k$
E.g. if $c = 2^{**}15$, i.e. 32768, loop takes at most $2*15 + 1$ iterations!