Correctness Issues

Murali P

Correctness Issues

- Motivation
- Modules, Contracts, Invariants
- Tests and Test Cases

What is correctness?

- Design Correctness
  - Solution (design) meets requirements
  - Verified offline (often on paper)
  - Proof arguments
- Implementation Correctness
  - Implementation (program code) matches design
  - Verified online (often by execution)
  - Tests and Test Cases

What is Correctness?

- Output of Design step: Program Design
  - High level solution to problem
  - Consists of modules and module interconnections
  - Modules are solutions to sub-problems
  - Interconnections capture ways to combine sub-solutions

Design Correctness

- Design Correctness involves
  - Module correctness (for each module)
  - Combination correctness
- Design Correctness is verified by correctness arguments:
  - Establish Module Correctness
  - Verify contracts between modules.

Module Correctness

- Often reduces to algorithm correctness:
  - Algorithm will terminate
  - Algorithm will produce required result if and when it terminates.
- Both arguments are fairly easy for “straight-line” programs – i.e., no loops.

How the post conditions are met?

Example: 1

/* pre condition: x < 2
post condition x< 10 */

```
int compute (int x)
{
/*  pre-condition of S1: x < 2 */
/*S1*/      int y = 3*x+1;
/*  post-condition of S1: x < 2, y < 7 */
/*  pre-condition of S2: y < 7 */
/*S2*/      x= y+3;
/*  post-condition of S2: x < 10 */
      return x;
}
```

/* post-condition of **compute**: x < 10 */

Example: 1

int compute (int x)
{

/*S1*/ int y =3*x+1; [x<2]
i.e., $y < 3*2 + 1 \Rightarrow y < 7$

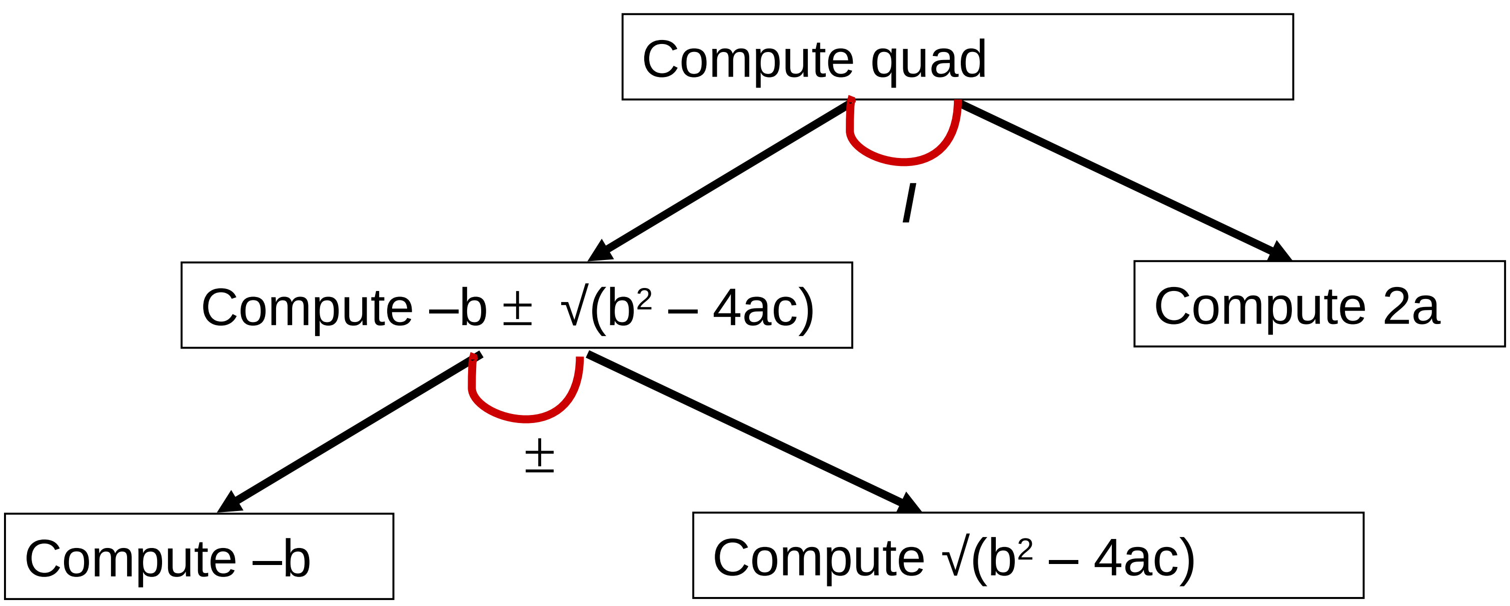
/*S2*/ x=y+3; [y<7]
$x < 7+3; \Rightarrow x < 10$

/*S3*/ return x;
}

Module Correctness

- Problem: Given a, b, and c, solve quadratic equation
  –

  $a*x^2 + b*x + c = 0$



Module Correctness

- Solution: Define function quad(a, b, c, sign) as

```
disc = b*b – 4*a*c;
if (sign)
   return (-b + sqrt(disc)) / (2
  *a);
else
   return (-b – sqrt(disc)) / (2*a);
```

Module Correctness

- Termination:
  - if sqrt terminates, this function terminates.
- Valid results:
  - if sqrt is correct then this returns correct value.
- How do we handle the “if” conditions above?
  - Contracts

Inter-Module Correctness

- Whoever writes sqrt function, specifies input-output contract:

/* Pre-condition: $m > 0$

Post-condition: return n such that

$$|n * n - m| / m < .01$$

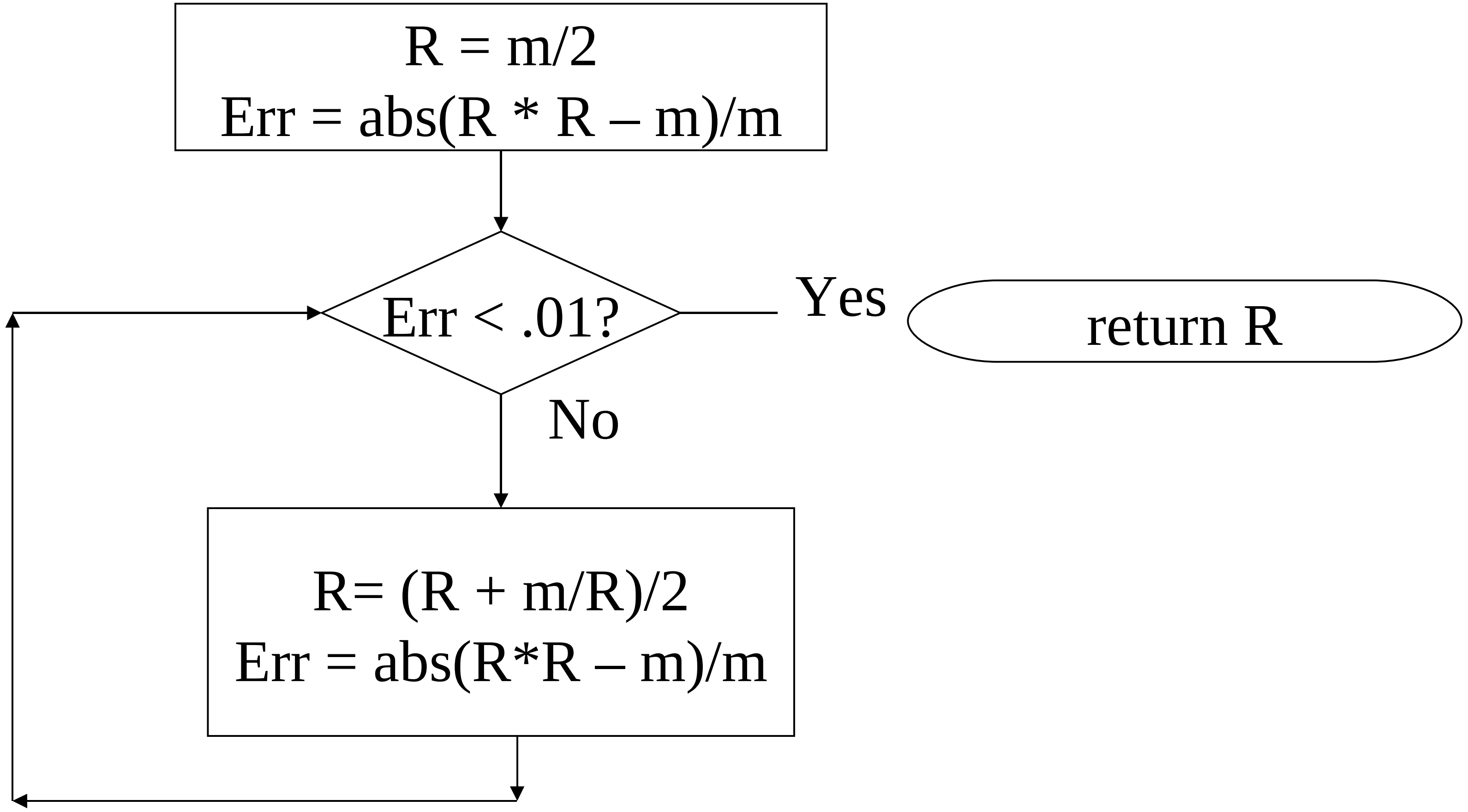
*/

float sqrt(float m) → argument must be +ve definite

{

}

Module Correctness for sqrt(x)



```
float sqrt (float m)
{
    float R=m/2;
    float Err= (R*R – m)/m;
    while (Err >= 0.01)
    {
        R= (R + m/R)/2;
        Err=(R*R – m)/m;
    }
    return R;
}
```

```
/*
Pre condition:
    m > 0
Post condition:
    R = sqrt(m) (with a relative
    error less than 1%)
    Err < 0.01
*/
```

Inter-Module Correctness

- Observation:
  - Precondition $m > 0$
  - This is required for sqrt to be correct (or may be even to terminate).
  - So, quad module must guarantee before invocation of sqrt:

    $disc > 0$

Inter-Module Correctness

- The previous contract may propagate up:

/* Pre-condition: b*b > 4*a*c

Post-condition: return x such that

| a*x*x + b*x + c | <= epsilon

*/

float quad(float a, float b, float c, int sign)

{

}

- Function interfacing errors minimized due to the
  - Pre conditions and Post conditions.
- Why do need correctness ?
  - Ensure “logic” is correct
  - Testing will be easy
  - Helpful to the third party users
- Correct way a writing a function from now on:

/* **Pre-condition : ……. */**

/* **Post-condition: …….***/

<Function >

Correctness of Algorithms

- After writing a program,
  - test on sets of sample data.
- Choose sample data to test the correctness in extreme cases.
- Testing on sample data can't give perfect confidence that the program is correct.
- Need a proof that the program is correct.
- Many methods of proofs for program correctness are based on induction.

Pre-conditions and Post-conditions

- The predicate describing the initial state is called the pre-condition of the algorithm.
- The predicate describing the final state is called the post-condition of the algorithm.
- **Example:**

In an algorithm which computes

the product of prime numbers $p_1, p_2, \ldots, p_n$:

pre-condition: The input variables $p_1, p_2, \ldots, p_n$ are prime numbers.

post-condition: The output variable q equals $p_1 \cdot p_2 \cdot \ldots \cdot p_n$.

How the post conditions are met? - A revisit to single-step routines

- Take the pre condition, verify one step at a time
- Infer the post-condition

Example 2: Swap 2 Numbers

/*Precondition: x and y are integer values
Post condition: x and y swapped
Example x=6, y=8 */

```
void swap (int x, int y)
{
  printf(“x=%d y=%d\n”,x,y);
   x=x+y;
   y=x-y;
   x=x-y;
  printf(“x=%d y=%d\n”,x,y);
}
```

x=6, y=8

y=8, x=x+8 ⇒ x= 14

x=14, y=x-8 ⇒ y=6

y=6 , x=x-6 ⇒ x= 8

Post Condition: values are interchanged

How the post conditions are met?
- A revisit to single-step routines

- Take the post condition, push it up, one step at a time
- Infer the pre-condition

Example 2: Swap 2 Numbers

/*Precondition: x and y are integer values
Post condition: x and y swapped
Example x=6, y=8 */

```
void swap (int x, int y)
{
  printf(“x=%d y=%d\n”,x,y);
   x=x+y;
   y=x-y;
   x=x-y;
  printf(“x=%d y=%d\n”,x,y);
}
```

x - 8=x’ ⇒ x=6, y=8
14 - y=y’ ⇒ y=8, x=14
x+6=x’ ⇒ x=14, y=6
x=8, y=6

Correctness for Conditional statements

Example 1:

//Post condition of if-else block: y >0

//Pre condition: ???

```
int fun1(int x, int y)
{
  if(x >0)
      y = y-1;
  else
      y = y+1;
```

//Precondition for next statement: y >0

```
  return sqrt(y);
}
```

Correctness for Conditional statements-

Example 2: Largest of 3 numbers

```
int large3(int a, int b, int c)
{
    int result;
    if((a>=b) && (a>=c))
        result =a;
    else if ((b>=a) && (b>=c))
        result =b;
    else
        result =c;
    return result;
}
// Precondition: a, b, and c are integer values
//Post condition: result is the largest of a, b and c
```

Loop Invariants: Method to prove correctness of loops

- Loop has the following appearance:

  [pre-condition for loop]

  while (Guard)

  [Statements in body of loop. None contain branching statements that lead outside the loop.]

  end while

  [post-condition for loop]

Loop-Invariant

- Every loop has a pre-condition and post-condition
- There is some condition G
  - True ➔ the loop will continue and
  - False ➔ the loop terminates
- Some condition I remains constant in loop
  - Correct before loop begins
  - In each iteration of loop
  - After loop terminates

Loop Invariance

```
/* Pre –condition :    N >=0 */
/* Post –condition: fact =N! and i>0*/
    int factorial (int N)
    {
       int i, fact =1;
     for( i=1; i<=N; i++)
            fact = fact * i;
       return fact;
     }
```

```
/* Pre –condition :    N >=0 */
/* Post –condition: fact =N! and i>0*/
int factorial (int N)
    {
        int fact =1; int i=1;
         /* fact = i-1! and i>0 */
        while (i<=N)
         {
              fact = fact * i;
                i++;
              /* fact = i-1! and i>0 */
         }
             /* fact = i-1! and i>0 */
         return fact;
     }
```

Loop Invariant: fact=i-1! and i>0 and i<=(N+1)

Loop Invariance

// Precondition: N>=0

// Post condition: fact = N! and i>=0

Loop Invariant: N!=fact * i! and i>=0 and i<=N

```
int factorial (int N)
    {
        int fact =1; int i=N;
        /* N! = fact * i! and i>=0 */
        while (i>0)
        {
            fact = fact * i;
            i – – ;
          /* N! = fact * i! and i>=0 */
        }
          /* N! = fact * i! and i>=0 */
        return fact;
    }
```

Loop Invariants: Method to prove correctness of loops

- Loop has the following appearance:

  [pre-condition for loop]

  while (Guard)

  [Statements in body of loop. None contain branching statements that lead outside the loop.]

  end while

  [post-condition for loop]

Loop-Invariant

- Every loop has a pre-condition and post-condition
- There is some condition G
  - True ➔ the loop will continue and
  - False ➔ the loop terminates
- Some condition I remains constant in loop
  - Correct before loop begins
  - In each iteration of loop
  - After loop terminates

Loop Invariance Example 1

```
/* Pre –condition :    N >=0 */
/* Post –condition: fact =N! and i>0*/
int factorial (int N)
    {
        int fact =1; int i=1;

        /* fact = i-1! and i>0 */
        while (i<=N)
         {
               fact = i;
               i++;
          }

         return fact;
     }
```

Loop Invariant: fact=i-1! and i>0

What is a loop invariant?

- Property that is maintained "invariant" by iterations in a loop.
  - Captures progressive computational role of the loop and remaining true before and after the loop irrespective of how many times the loop is executed
- How is it used:
  - Verify before the loop
  - Verify each iteration preserves it.
  - Property on termination of loop must result in "what is expected"

Using Loop Invariance

- Verify the following properties

1) ***Basis property:*** The pre-condition implies that I is true before the first iteration of the loop
   - fact = i-1! and i>0

2) ***Inductive property:*** **If** guard **G and I** are true before an iteration, **then I** is true after the iteration.
   - fact = i-1! and i>0 is preserved by the body of the loop

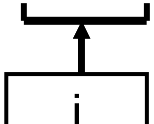
3) ***Eventual falsity of Guard:*** After finite number of iterations of the loop, the guard G becomes false. [Termination condition]
   - i eventually becomes > N

1) ***Correctness of the Post-Condition:*** **Q: (NOT G) and I**

   implies the post-condition. **{Q} ={I and ! G }**

   when termination occurs, i.e., i=N+1;

   fact=((N+1)-1)! which is same as post-condition fact=N!



Loop Invariance Example 1_Part 2

// Precondition: N>=0

// Post condition: fact = N! and i>=0

Loop Invariant:

N!=fact * i! and i>=0

```
int factorial (int N)
    {
        int fact =1; int i=N;
        /* N! = fact * i! and i>=0 */
        while (i>0)
        {
            fact = fact * i;

            /* N! = fact * i! and i>=0 */


         /* N! = fact * i! and i>=0 */
         return fact;
    }
```

Loop Invariance – Example 3

- Consider the following Alg to find the index of largest integer in a list of integers

```
/* Pre condition: list(N) is assigned N positive integers
   Post condition: list[indexMax] is a maximum element in list
*/
int maxIndex( int A[ ], int N)
{       int k, indexMax;
        indexMax = 0; k=1;
        while(k < N)
        {
                if (A[k] > A[indexMax])
                        indexMax = k;
                k++;
         }
        return indexMax;
}
```

- Correctness:

  $\forall$ j, 0 <= j < N, A[j] <= A[indexMax] and 0 <= indexMax < N

- A useful invariant for the above loop:

  $\forall$j: 0 <= j < k, A[j] <= A[indexMax] and 0 <= indexMax < k

Loop Invariant – Example

- *Basis property*:

  When k = 1: indexMax = 0, and j=0 only

  A[j] = A[0]=A[indexMax] and indexMax=0<k.

- *Inductive property*:

  Assume that the invariant is true for k = k0.

  If A[k0] <= A[indexMax], then the invariant remains true for the new k = k0+1, after executing k++

  If A[k0] > A[indexMax], then after executing k++, indexMax = k0 and A[j] <= A[indexMax], j <k0+1, Since indexMax < k0+1, this verifies the invariant for k = k0+1.

Loop Invariant – Example

- *Eventual falsity of Guard: simple to prove in this case*
- *Correctness of the Post-Condition: Falsity of the guard and invariant implies the post condition*

  (k = N) and (for all j, 0 <= j < k, A[j] <= A[indexMax], and 0 <= indexMax < k)

Loop Invariance Example 4

- For the algorithm to find x^y, (discussed earlier)
  - Write the module correctness (pre, post conditions)
  - Find an appropriate invariant
  - Discuss the correctness argument

Correctness: Second Example

- Algorithm for x^y
  - Extract a power of 2 from y, say P.
  - Compute x^P and multiply this to a temp. result
  - Repeat above steps until nothing to extract.

Correctness: Second Example

- Algorithm for x^y

```
Ynext = y; Power = 1; Result=1;
while (Ynext > 0) do
  if (Ynext mod 2 == 1)
  then Result = Result * pow2(x, Power);
  Power = 2 * Power;
  Ynext = Ynext / 2;
endwhile;
```

Module Correctness: x^y

- Loop Invariant:

  $x^y = R * x^{(P*Ynext)}$

- Before the loop:

  $x^y = 1 * x^{(1*y)}$

- Inside the loop:

  $x^{(P*Ynext)} = x^{(2*P*(Ynext/2))}$ if Ynext is even;

  $x^{(P*Ynext)} = x^{(2*P*(Ynext/2))+P}$

  $= x^{(2*P*(Ynext./2))} * x^P$ if Ynext is odd;

Module Correctness: x^y

- Termination:

  Ynext is reduced by (at the least) half for each iteration.

  So, for positive y, Ynext will eventually be 0 –

  because of integer division.

- Pre-condition for Module x^y:

  $y >= 0$

Inter-Module Correctness

- Assumption:
  - pow2(x,P) returns x^P if P is a power of 2.
- Definition of pow2

```
/*  Pre-condition: P = 2^k for some k >= 0
      Post-condition:  return x^P  */
int pow2(int x, int P)
…
```

Loop Invariance Example 5

Binary Search

/* Pre condition: N>0, A is in non-decreasing order

Post condition:

(1) x = A[m] and m is the location of the element

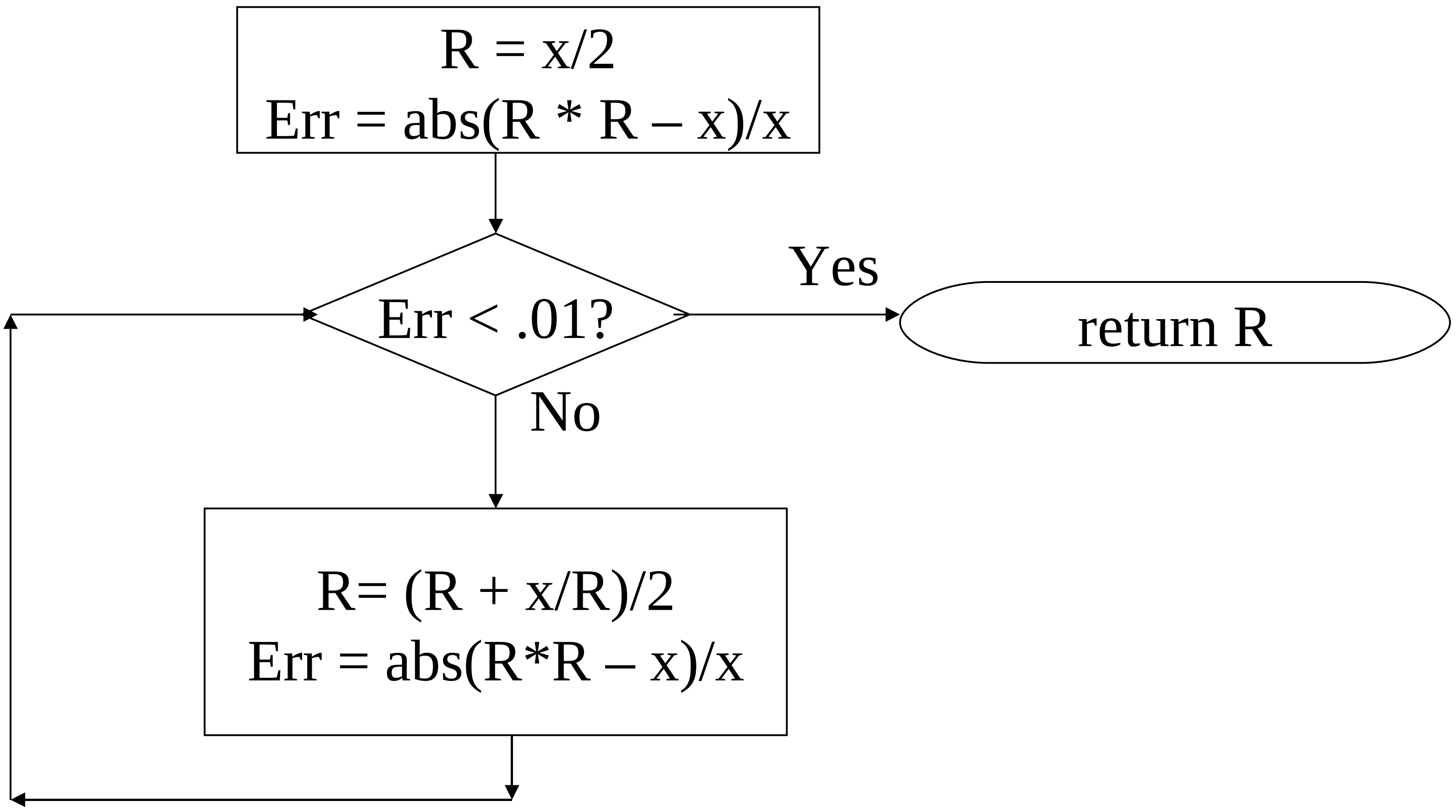
OR

(2) x in not in A

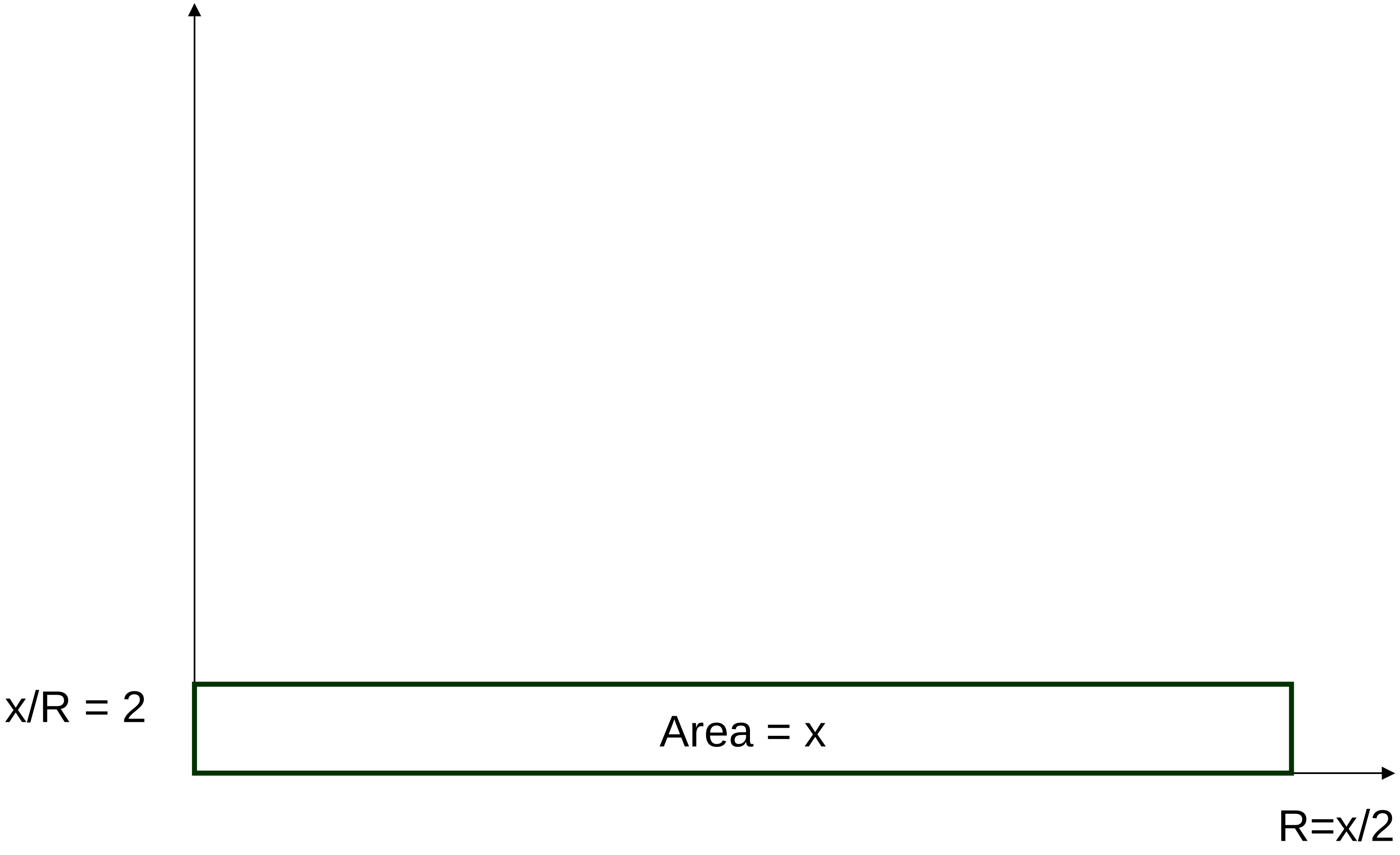
*/

```
int bsearch(int a[ ], int N, int x)
{
    int lo=0; int hi=N-1; int mid;
      Invariant: (x is not in A) OR (A[lo] <=x<=A[hi])
    while (lo <= hi)
    {
        mid = (lo + hi)/2;
        if (A[mid] == x) return mid;
        else if (A[mid]>x) hi=mid-1;
        else lo = mid+1;
    }
    return -1 // Not found in the given list A
}
```

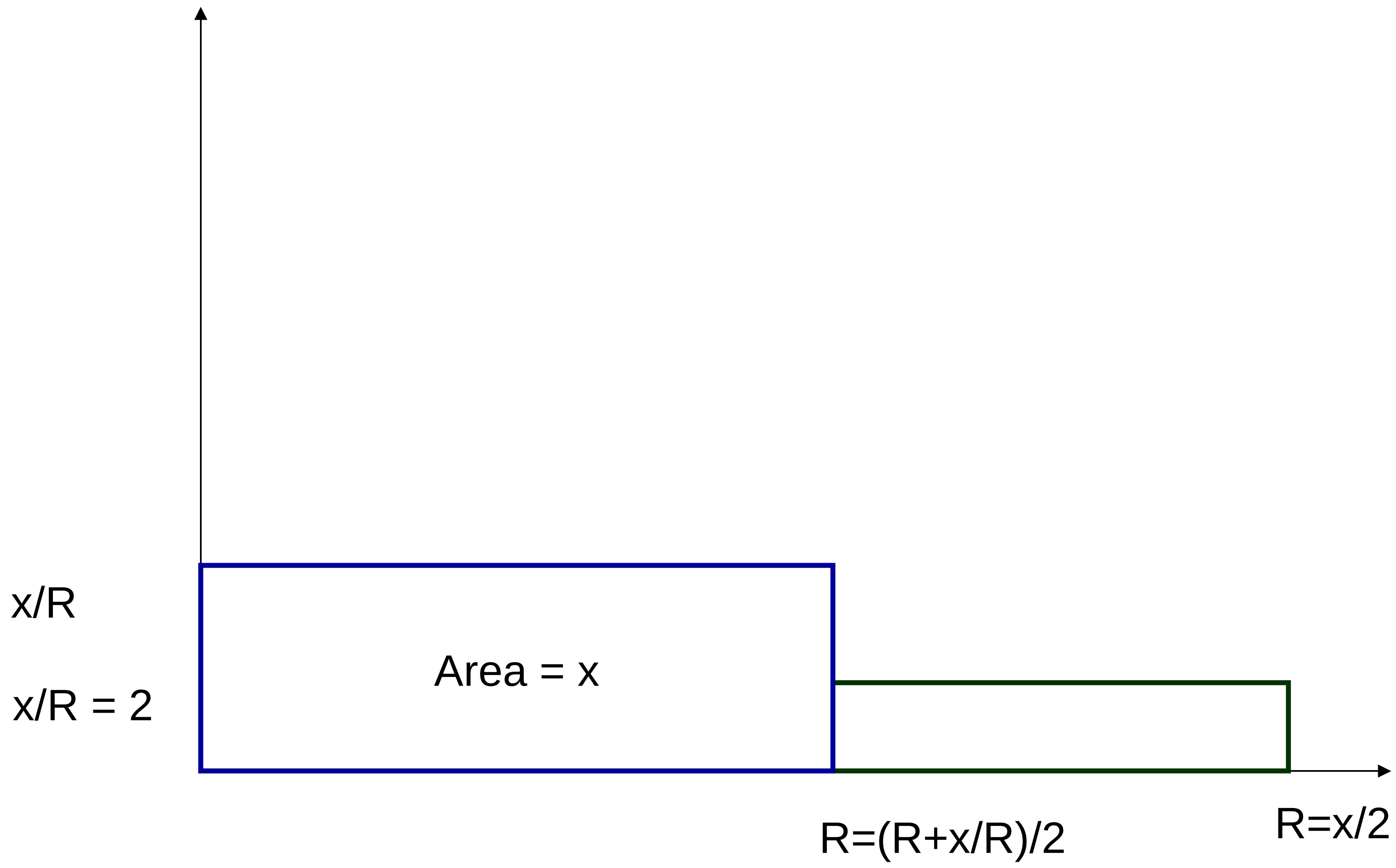
Module Correctness for sqrt(x)



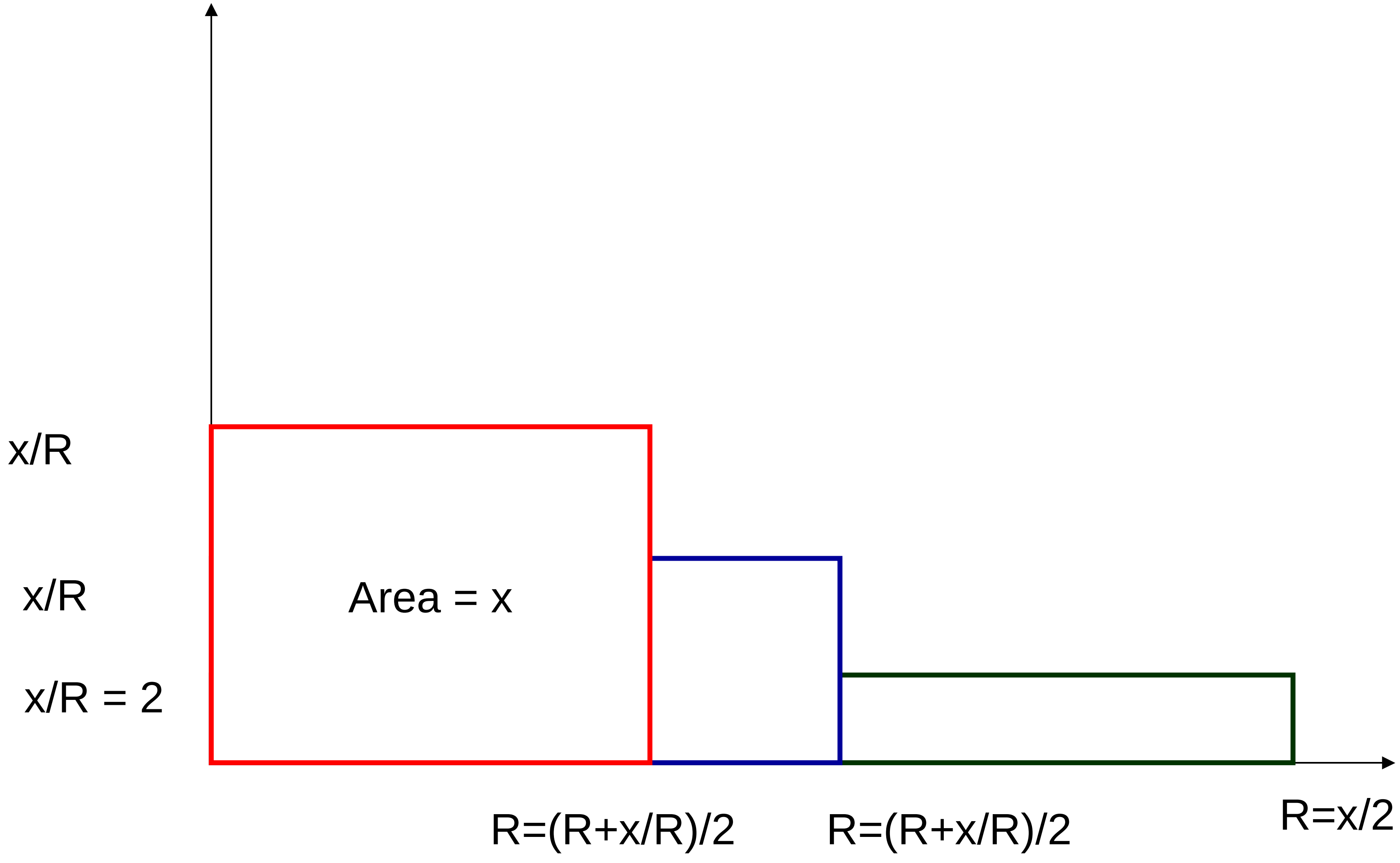
Square root



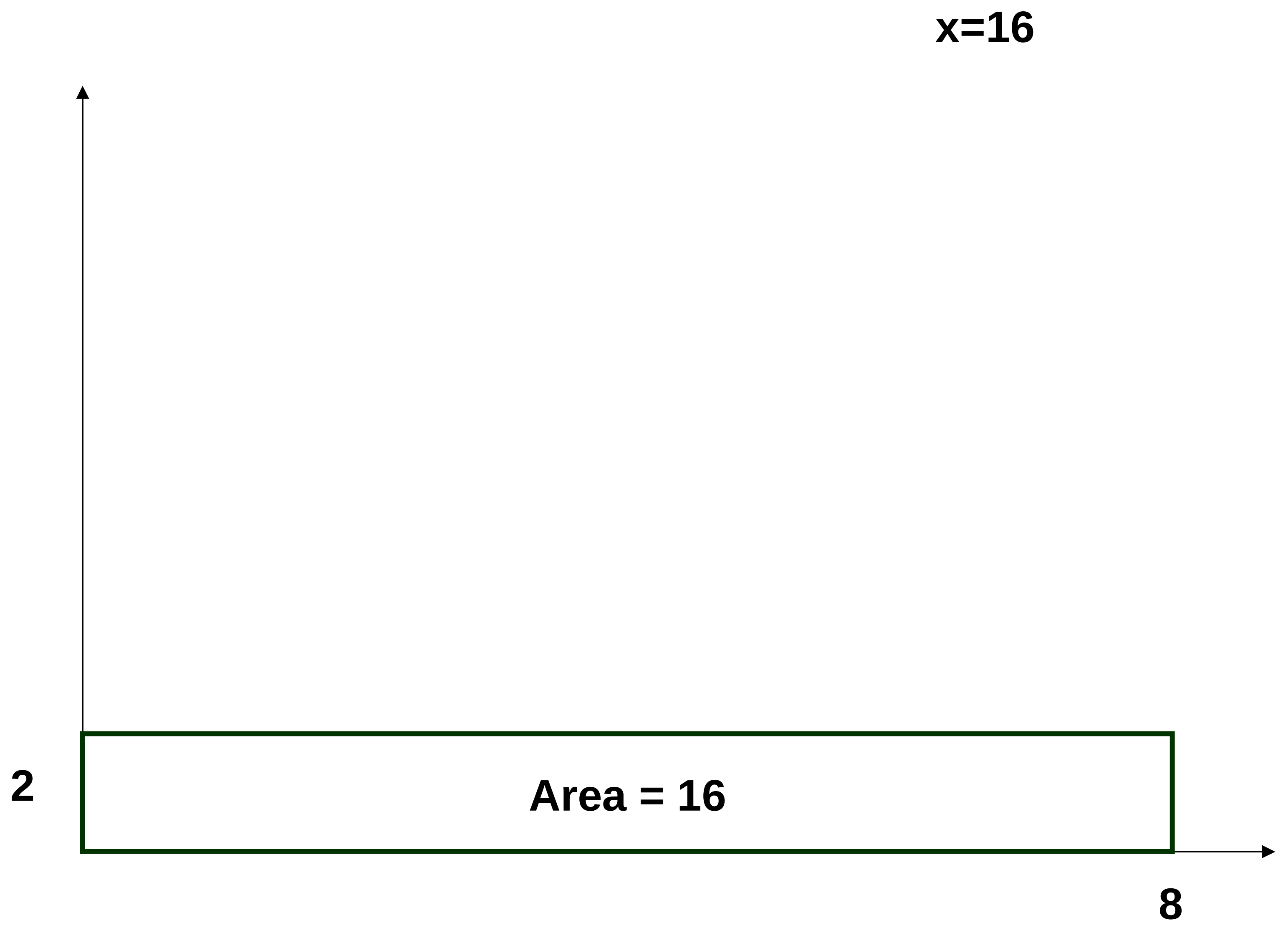
Square root



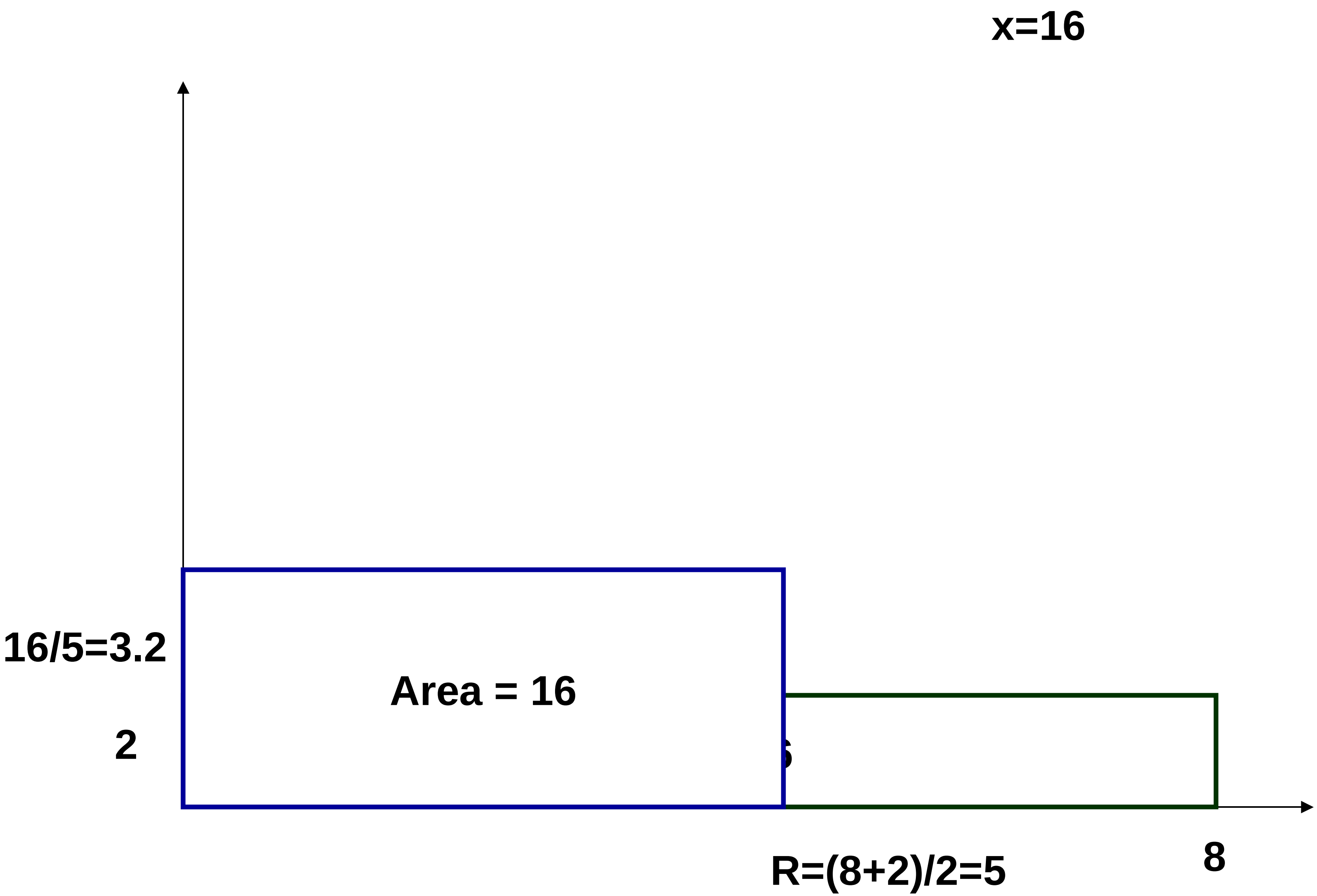
Square root



Square root

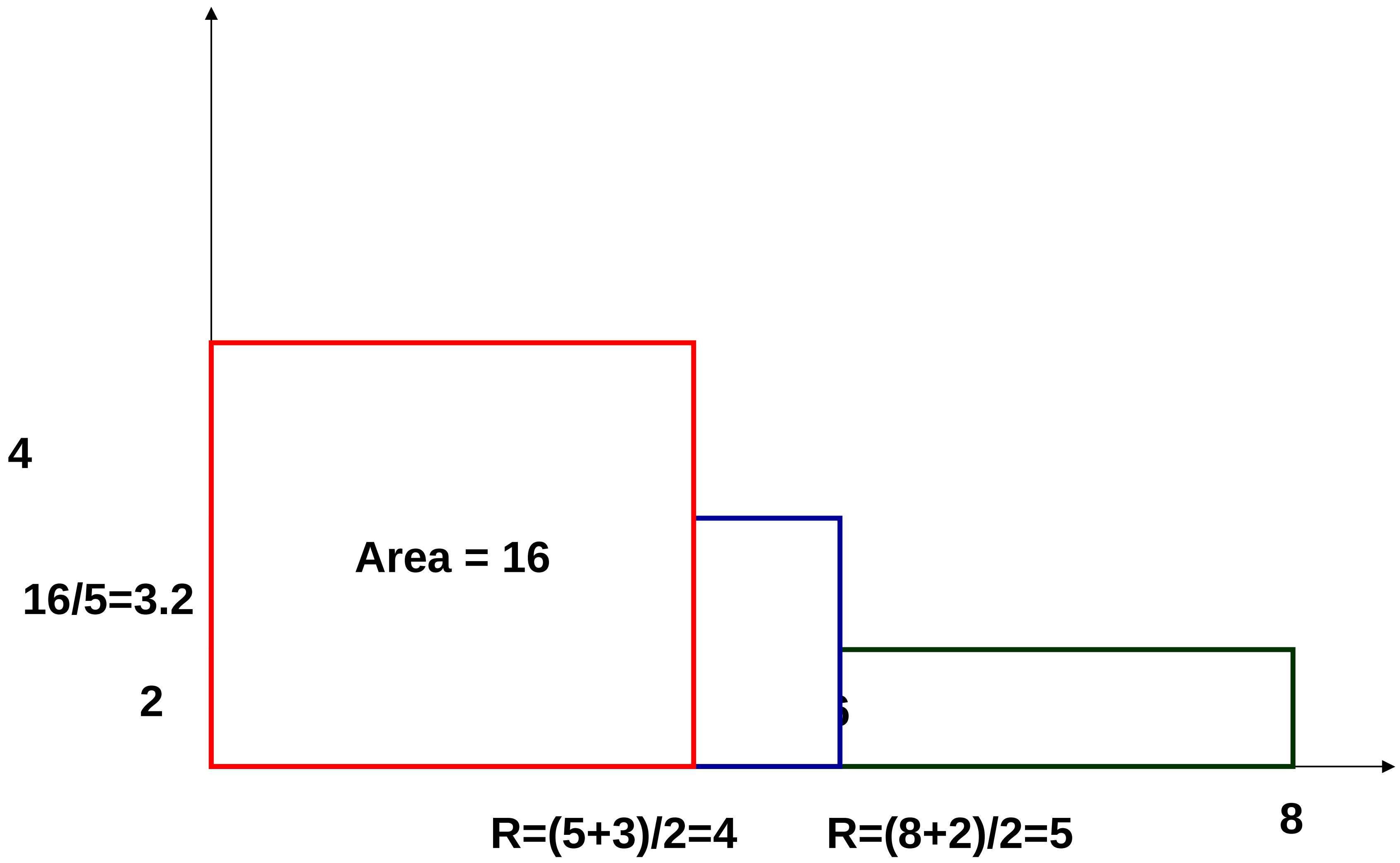


Square root



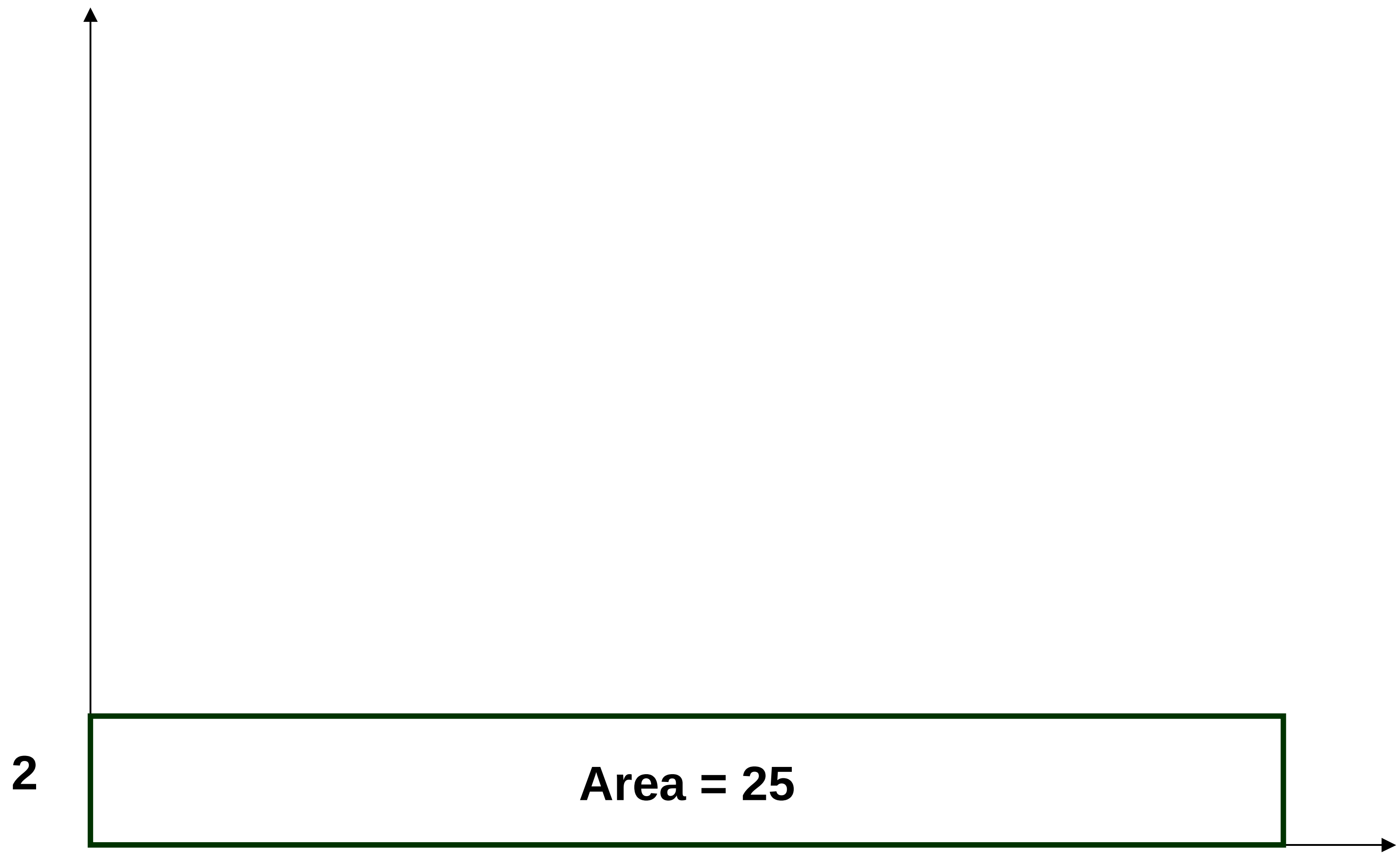
Square root

x=16



Square root

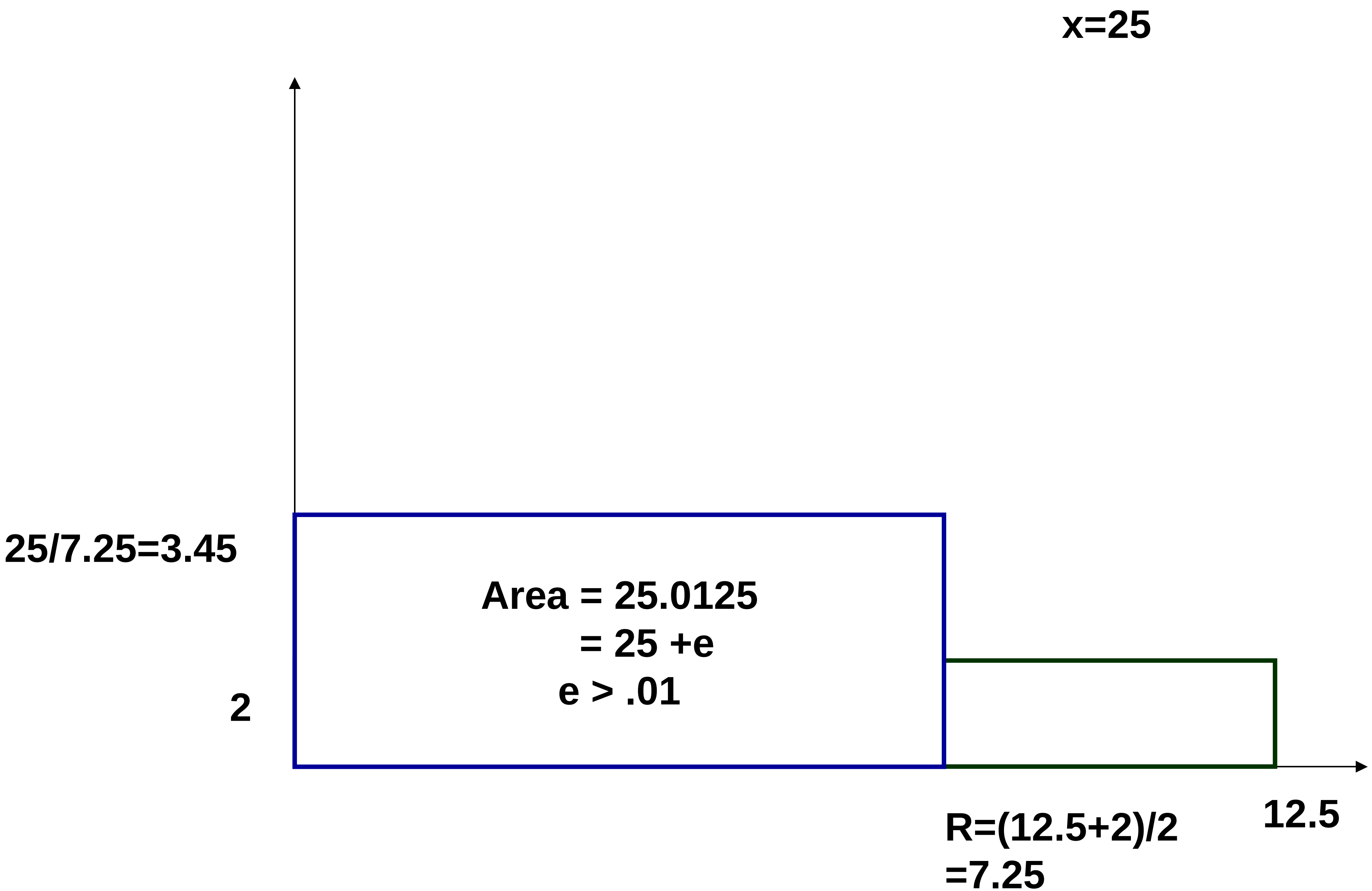
x=25



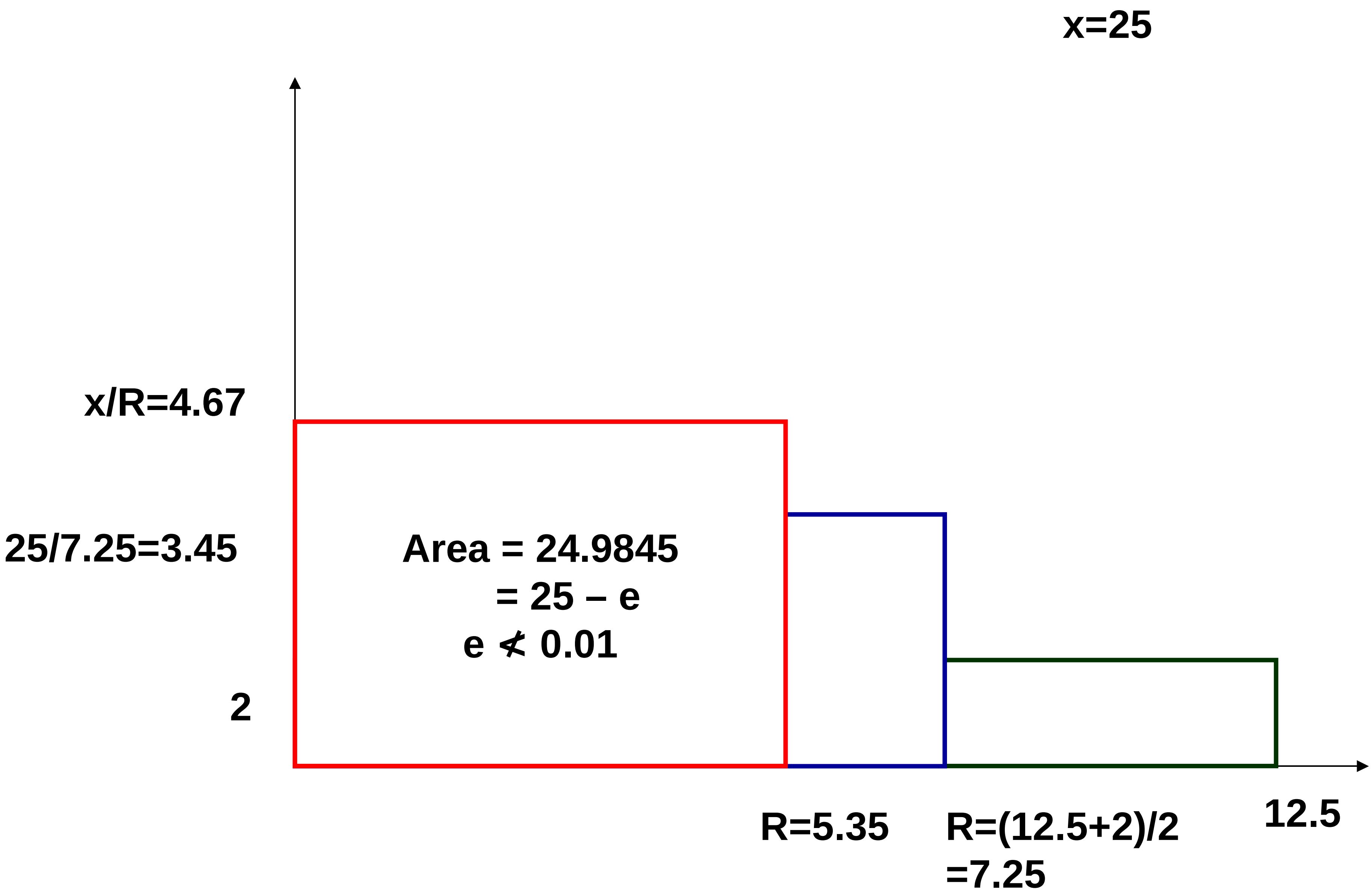
12.5

R=5.01 R=5.35 R=(12.5+2)/2
=7.25

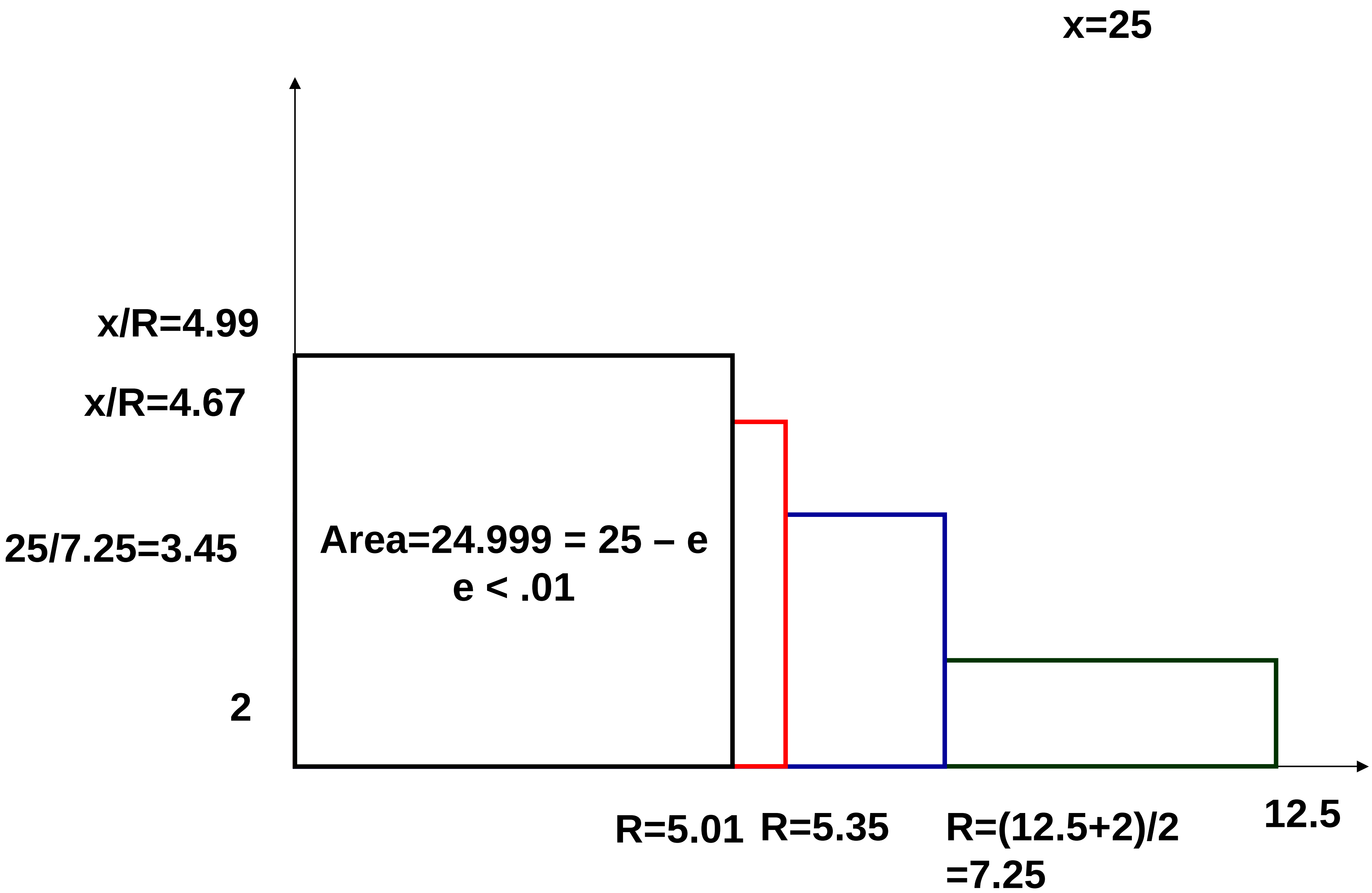
Square root



Square root


x=25
x/R=4.67
25/7.25=3.45
Area = 24.9845
= 25 – e
e ≮ 0.01
2
R=5.35
R=(12.5+2)/2
=7.25
12.5

Square root


x=25
x/R=4.99
x/R=4.67
25/7.25=3.45
Area=24.999 = 25 – e
e < .01
2
R=5.01
R=5.35
R=(12.5+2)/2
=7.25
12.5

Loop Invariance Example

```
/*
Pre condition:
    m > 0
Post condition:
    R = sqrt(m) (with a relative
    error less than 1%)
    Err < 0.01
*/
```

```
float sqrt (float m)
{
    float R=m/2;
    float err= (R*R – m)/m;
    while (fabs(err) >= 0.01)
    {
        R= (R + m/R)/2;
        err=(R*R – m)/m;
    }
    return R;
}
```

Module Correctness for sqrt(x)

- Termination?
  - Start with a range for R: x/2
  - Every step reduces the size of the range: average is closer to the middle than the ends
  - Range should get smaller and smaller – must terminate when R is close to the root.

Module Correctness for sqrt(x)

- Loop Invariant:

  R <= sqrt(x) <= x/R   OR   x/R <= sqrt(x) <= R

- Verify:
  - Universally true?
  - Side of a square vs. sides of a rectangle with same area.
  - At termination, (R*R – x)/x   is small (approx.)