Floor Square Root

Find a function Floor_Sqrt s.t.

```
{x ≥ 0.0}
r = floor_sqrt(x)
{r = ⌊ √ x ⌋ }
```

where

$\lfloor y \rfloor$ ("floor(y)") = greatest integer $\leq y$

e.g. $\lfloor 3.14 \rfloor = 3$ and $\lfloor -3.14 \rfloor = -4$

Alternative Defn. $\lfloor x \rfloor$ **"floor(x)"**

$$n = \lfloor x \rfloor \equiv n \leq x < n+1$$

e.g. $r = \lfloor \sqrt{x} \rfloor \equiv r \leq \sqrt{x} < r+1$

$\equiv r^2 \leq x < (r+1)^2$

e.g $14 = \lfloor \sqrt{200} \rfloor$ as

$14^2 \leq 200 < 15^2$

i.e. $196 \leq 200 < 225$

Find Floor_Sqrt s.t.

$\{ x \geq 0.0 \}$
r = floor_sqrt(x)
$\{ r^2 \leq x < (r+1)^2 \}$

i.e.

Given a real number x, find an integer, r, s.t. $r^2 \leq x < (r+1)^2$.

e.g. let x = 200, then $14^2 \leq 200 < 15^2$

Note

To find $\sqrt{x}$ to n decimal places:

multiply x by 10^{2n}

Get Floor Square Root;

Divide the result by 10^n.

e.g. If x = 2 then Floor_Sqrt(100 * x) / 10 gets $\sqrt{2}$ to one decimal place.

Floor_Sqrt(200)/10 = 14/10 = 1.4 = $\sqrt{2}$ to one decimal place.

Linear Search for Floor Square Root

Use a Liner Search technique:

Starting with r = 0, iterate r until $(r+1)^2 > x$ we get

```
int  linear_sqrt(double x)
{    // precondition
     assert( x >= 0);
      int r = 0;
      while ( (r+1)*(r+1) <= x )
          r = r + 1;
      return r;
  //post:   r² ≤ x < (r+1)²
} // linear_sqrt
```

Starting with r = 0,

we increment r until we find the first or minimum, r , such that $(r+1)^2 > x$.

e.g. linear_sqrt(5.0) = 2 as $(2+1)^2 > 5$ i.e. $3^2 > 5$.

We have $2^2 \leq 5 < 3^2$, i.e. $4 \leq 5 < 9$.

e.g. we have $14^2 \leq 200 < 15^2$ i.e. $196 \leq 200 < 225$ therefore $14 \leq \lfloor\sqrt{200}\rfloor < 15$

Example: linear_sqrt(200)

$0^2 < 200$, $1^2 < 200$, …, $10^2 < 200$, …, $14^2 < 200$ as $14^2 = 196$ but $15^2 > 200$ as $15*15 = 225$.

Since $14^2 \leq 200 < 15^2$,

i.e. $14 \leq \sqrt{200} < 15$

therefore

linear_sqrt(200) = 14.

Alternative Program via Odd numbers

By induction it can be shown that the sum of the first n odd numbers is n^2

$$\sum_{k=1}^{n} (2 * k - 1) = n^2$$

Other Notation:

(+ k | 1 ≤ k ≤ n : 2*k-1)

e.g. (+ k | 1 ≤ k ≤ 5 : 2*k-1) = 1+3+5+7+9

= 25

i.e.

$$\sum_{k=1}^{5} (2 * k - 1) = 1 + 3 + 5 + 7 + 9 = 5^2 = 25$$

By summing odd numbers until sum greater than x we can get floor_sqrt(x) by:

```
int floor_sqrt(double x)
{
        int r, n, s;
        r = 0;
        n = 1;
        s = 1;
        while ( s <= x )
        {
             r = r + 1;
             n = n + 2;
             s = s + n;
        }
        return r;
} // floor_sqrt
```

In this program only the operation of addition is needed to find the floor square root of x.

Newton Method for Finding Square Root

The Newton Method for finding the roots of a diffrentiable function, f(x), is based on the following iterative process.

$$t_0 = 1$$

$$t_{k+1} = t_k - \frac{f(t_k)}{f'(t_k)}$$, where $f'(x)$ is the derivative of $f(x)$.

To find the square root of a real number, n, we consider

$$f(x) = x^2 - n$$

so that when f(r) = 0, we have

$$r^2 - n = 0$$

i.e. $r^2 = n$

i.e. $r = \sqrt{n}$

Using the Newton iterative process, with $f(x) = x^2 - n$

$$t_0 = 1$$

$$t_{k+1} = \frac{t_k^2 + n}{2t_k}$$

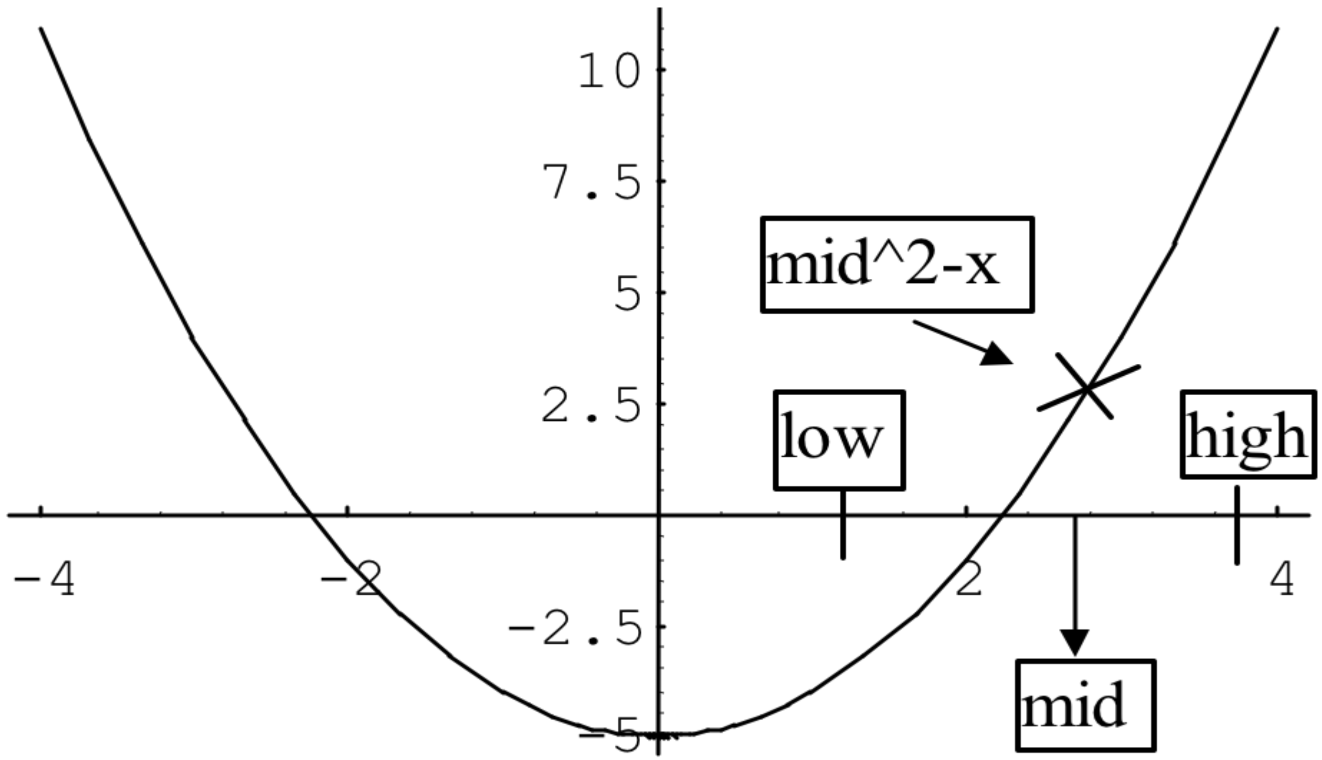
```
 double newton_sqrt(double n)
{
   double r = 1.0;              //  initial value
   double tiny = 0.0001;    // tolerance value

   while ( Math.abs((n - r*r)) > tiny )
       r = (r*r + n) / (2*r);
   return r;
} // newton_sqrt
```

Finding Square Root by Binary Search

To find the square root of n, find (an approximation of) the root of x^2 - n

e.g. n = 5; Find (+) solution to $x^2 - 5 = 0$.



```
double binary_sqrt(double low, double high, double tol,  double n)
{
    double mid;
    while ( low + tol < high )
    {
        mid = (low + high) / 2.0;
        if ( mid*mid - n  <=  0.0 )
            low = mid;
        else
            high = mid;
    }
    return low;
} // binary_sqrt
```

Comment:

The root, $\sqrt{n}$, lies between `low` and `high` i.e. `low` $\leq \sqrt{n} <$ `high` tf. $low^2 \leq n < high^2$.

We split the interval `(low, high)` and find which half the root is in,

e.g. let $mid = \frac{low+high}{2}$ then if $mid^2 - n > 0$ then root is in 'left' half and we reset `high` to be `mid` (see diagram).

More generally, in looking for `r`, such that `f(r) = 0`,

if `f(mid) > 0` then reset `high` to be `mid`

and if `f(mid) < 0` then reset `low` to be `mid`.

Since exact equality of Real numbers is not implementable, we use 'approximate equality'

i.e. for Real numbers x, y , we regard $x \approx y$ if x and y differ by a very small value,

i.e. `|x-y| < tol`, where `tol` is a very small tolerance value.

Each time through the while-loop, we have $low^2 \leq n < high^2$.

When the while-loop halts, we have

`high ≤ low + tol`, where `tol` is a small tolerance value.

Also, $low^2 \leq n < high^2$,

i.e. $\texttt{low} \le \sqrt{\texttt{n}} < \texttt{high}$

At termination we get

$$\texttt{low} \le \sqrt{\texttt{n}} < \texttt{high} \le \texttt{low} + \texttt{tol}$$

i.e. $\texttt{low} \le \sqrt{\texttt{n}} < \texttt{low} + \texttt{tol}.$

i.e. $\texttt{low} \approx \sqrt{\texttt{n}}.$

Picking initial interval: (low, high)

Let `low = 0;`

`high = n+1;`

then

$low^2 \leq n < high^2$

e.g. n= 10,000 tf. $\sqrt{n} = 100$

This initialisation gives an initial interval (0, 10,001)

Alternative:

Consider a smaller initial interval by finding least power of 2 greater that $\sqrt{n}$,

i.e. least $2^k > \sqrt{n}$

e.g. x = 10,000 tf. $\sqrt{n} = 100$

Alternative gives initial interval (0, 128).

```
double sqrt_bin(double n)
{
    double y = 1.0;
    while ( y*y <= n )
        y = 2.0*y;
    return binary_sqrt(0, y, 0.01, n);
} // sqrt_bin
```

Finding the Root of a Polynomial

Assume we have a function, eval(p, x) which evaluates a polynomial, p, at a value x.
A polynomial is defined by its co-efficients which may be stored in an array.

```
double root_poly(double low, double high, double tol,  double [ ] poly)
 {   // eval(poly, low) ≤  0 < eval(poly, high)
    //  The polynomial, poly, is monotonic increasing in [low..high]
     double mid;
     while ( low + tol < high )
     {
         mid = (low + high) / 2.0;
         if eval(poly, mid) < 0.0 )
              low = mid;
         else
              high = mid;
     }
     return low;
 } // root_poly
```

The function call, eval(poly, mid) evaluates the polynomial, poly, at the value, mid.