```c
int fib(int n) {
    if (n < 2)
        return n;
    else
        return fib(n-1) + fib(n-2);
}
```

The diagram illustrates the recursive calls for `fib(5)` and `fib(3)` with the calculated values:

- `fib(5) = 5`
- `fib(3) = 2` (values are 0, 1, 1, 2, 3, 5)

The text notes that the recursive approach is inefficient due to repeated calculations.
More efficiently:

```c
int fib(int n) {
    if (n < 2) return n;
    return fibHelper(n, 1, 1, 0);
}
```

```c
int fibHelper(int n, int k, int fibk, int fibk_1) {
    if (k == n) return fibk;
    return fibHelper(n, k+1, fibk+fibk_1, fibk);
}
```

`fib(4)`

fibHelper (4, 1, 1, 0);
```
  " (4, 2, 1, 1);
  " (4, 3, 2, 1);
  " (4, 4, 3, 2);  ⇒3
```
\[ m \geq n > 0 \quad \text{GCD} \quad k \quad \text{s.t. } k \text{ is the largest int that divides } m + n \]

Algorithm 0: Brute Force — try values of \( k \) from \( n \) down

```c
int gcd(int m, int n) { \text{\textit{initial guess}}
    return test(m, n, n);
}

int test(int m, int n, int guess) {
    if (m \% guess == 0 && n \% guess == 0)
        return guess;
    else
        return test(m, n, guess-1);
}

gcd(6, 4) \quad test(6, 4, 4) \quad test(6, 4, 3) \quad test(6, 4, 2) \quad \Rightarrow 2
```
Algorithm 2: Euclid’s algorithm

for \( m \geq n > 0 \), \( \text{GCD}(m,n) = \) \( \begin{cases} n & \text{if } m \% n = 0 \\ \text{GCD}(n, \text{remainder of } m/n) & \text{otherwise} \end{cases} \)

Why? We can rewrite \( m \) as:

\[
m = n \left( \left\lfloor \frac{m}{n} \right\rfloor \right) + \text{remainder of } \frac{m}{n}
\]

So this must also be divisible by \( d \)!!
for \( m \geq n > 0 \), \( \text{GCD}(m, n) = \begin{cases} n & \text{if } m \% n = 0 \\ \text{GCD}(n, \text{remainder of } \frac{m}{n}) & \text{otherwise} \end{cases} \)

```c
int gcd(int m, int n) { 
    if (m % n == 0) 
        return n;
    else
        return gcd(n, m % n);
}
```

gcd(48, 24) = gcd(24, 12) = gcd(12, 12) = gcd(12, 0) = 12

gcd(135, 19) = gcd(19, 2) = gcd(2, 1) = 1
Algorithm 3: Dijkstra's Algorithm

Not allowed to use division or mod(%) 

Idea: If $m > n$, $\text{GCD}(m,n) = \text{GCD}(m-n,n)$

Why? If $\frac{m}{d}$ and $\frac{n}{d}$ are integers, then $\frac{m-n}{d}$ is an int.

For $m, n > 0$ 

$$\text{GCD}(m,n) = \begin{cases} 
  m & \text{if } m = n \\
  \text{GCD}(m-n,n) & \text{if } m > n \\
  \text{GCD}(m, n-m) & \text{if } n > m 
\end{cases}$$
\[
gcd(468, 24) \\
gcd(444, 24) \\
gcd(420, 24) \\
\vdots \\
gcd(36, 24) \\
gcd(12, 24) \\
gcd(12, 12) \\
\Rightarrow 12
\]
Problem: $x > 0 \rightarrow \sqrt{x} \rightarrow \sqrt{x}$

Rewrite: $k$ s.t. $k \cdot k = x$

$\Rightarrow k = \frac{x}{k}$

$\sqrt{11}$

$[9, 16] \Rightarrow [3, 4]$  

Guess 3.5, compare to $\frac{11}{3.5}$

3.5 $\approx$ 3.14

Average: 3.32 - next guess

3.32 comp. to $\frac{11}{3.32}$

3.32 $\approx$ 3.31

⇒ Good!
Algorithm:  
1. guess some value $g$ for $\sqrt{x}$
2. compute $x/g$
3. if $x/g$ is close enough to $g$, return $g$
   otherwise, try a better guess by averaging $x/g$ and $g$. 
Algorithm:

1. guess some value $g$ for $\sqrt{x}$
2. compute $x/g$
3. if $x/g$ is close enough to $g$, return $g$
   otherwise, try a better guess by averaging $x/g$ and $g$.

true if $a + b$ are “close enough”

a number closer to $\sqrt{x}$ than $g$ is
double sqrt(double x) { // assume x > 0
    return test(x, 1);
}

double test(double x, double g) {
    if (closeEnough(x/g, g))
        return g;
    else
        return test(x, betterGuess(x, g));
}

double betterGuess(double x, double g) {
    return (g + x/g)/2;
}
CloseEnough?

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>8</td>
<td>NO</td>
</tr>
<tr>
<td>10</td>
<td>10.1</td>
<td>NO</td>
</tr>
<tr>
<td>10</td>
<td>10.05</td>
<td>✓</td>
</tr>
<tr>
<td>1,059,672</td>
<td>1,059,671</td>
<td>✓</td>
</tr>
<tr>
<td>0.000001</td>
<td>0.05</td>
<td></td>
</tr>
</tbody>
</table>

--> looks like we care about the ratio

Exercise: Write the closeEnough method.