Whole Root

A simplified approach to finding the square root of a number.

Problem Statement

Let's look at how you solve a fairly straightforward math problem. We're going to explore square roots.

Write a function that will take a number `x` and tell you what the square root of that number is.

Here's a twist to make the problem a bit simpler: only return the square root if it would be a whole number (0, 1, 2, and so on). If `x` doesn't have a whole number for it's square root then simply return `false`.

Here are some examples of what your function should return when we call it:

```
wholeRoot(4); // returns 2
wholeRoot(9); // returns 3
wholeRoot(13); // returns false
wholeRoot(null); // returns false
wholeRoot('candybar'); // returns false
wholeRoot(0); // returns 0
wholeRoot(-4); // returns false
```

Suitability

The math in this problem is primary school level. The simplification makes it straightforward to solve with a brute force solution. Several optimizations are possible.

Using basic math and requiring no special computer science approach makes this suitable for a broad range of candidates and shouldn't give an unfair advantage to any particular background.

Implementation Quirks

The most straightforward solution uses a loop to start `i` at `0` and compute every square on the way to `x`. If `i*i === x` then return `i`.

Candidates should eventually consider checking the type of `x` and immediately returning `false` when `x<0`.

Interview Waypoints

State the problem clearly and consistently every time using the **Problem Statement** above.

- Candidates commonly use a loop to brute force the solution and they should be able to identify this possibility on their own.
- Using a brute force technique may embarrass candidates. You might probe them on their thinking and get around to reassuring them that a brute force solution would be an acceptable starting point.
- Candidates may fail to check the input. Point them to the examples calling the function with strings, `null`, and negative numbers. See if they add checks to their solution.
- If the candidate completes a brute force solution, probe on how they might make the solution more efficient. They should be able to identify at least some of the following ideas:
  - What about exiting the loop as soon as `i*i > x`?
  - What about performing a binary search rather than incrementing by 1?
  - with the exception of 0, and 1, the whole root of a number is never greater than `x/2`. What about terminating when `i > x/2`?

Expanding Beyond JS

Once the candidate has defined the function you can briefly probe into their comfort with assembling web pages by asking them to expand their solution into a simple web page.

Problem Statement:

Given the function you wrote please put it in it's own file. Write the markup for an html page that when viewed in the browser shows two rows: row 1 shows an empty input, row 2 the word `false`. Link your function to the page and write the glue code to make it take the input from row 1 as `x` and display the whole root of `x` in row 2.

Finally, link a CSS file to the page. In the CSS file write a rule that causes the result in row 2 to be green when it's a number and red when it's `false`.

Implementation Quirks

Expanding the problem into a simple web page gives you the chance to see how comfortable the applicant is making a complete web page. Do they understand the mechanics? Having the syntax memorized isn't necessary, but do they know the kinds of tags and attributes required?

You might ask them how they would go about finding the proper syntax. Where would they get their boilerplate from?

Conclusion

The whole root problem isn't mathematically intense. It involves enough fundamental concepts to give you a peek into the candidate's coding style and speed.