Can Local LLMs Analyze Algorithmic Complexity?

Ivan Murashko

Contents

Introduction

Choosing a local language model is not only a question of tokens per second. For development work, it is also important to know whether the model can read a small piece of code and recover the right algorithmic invariant. This article uses one benchmark question for this purpose. The question is deliberately small: estimate the time complexity of a recursive integer-square-root style implementation.

The interesting part is that the numeric input x does not change in the recursive call. The value that changes is an internal state variable. Thus a model that writes a recurrence directly in terms of x usually gets the wrong answer. The correct analysis has to introduce the distance from the current search point to $\lfloor\sqrt{x}\rfloor$.

1 Benchmark Setup

The benchmark was a local measurement, not a public leaderboard. The main staged run used the following environment:

- MacBook Pro Mac16,5;
- Apple M4 Max;
- 16 CPU cores and 40 GPU cores;
- 128 GB memory;
- Ollama 0.21.2;
- deterministic generation settings: `temperature=0`, `num_ctx=8192`, no streaming, and the runner requested `think=false` where supported.

The later hint-sensitivity runs were made on the same local benchmark setup with Ollama 0.24.0. They are reported separately, because adding a hint changes the task.

2 The Code

The benchmark used the following C++ class:

```cpp
class Finder {
public:
    int function(int x) {
        return findNearest(0, x);
    }

private:
    int findNearest(unsigned int start, int x) {
        unsigned int k = 1;

        for (unsigned int i = start; i <= x; i += k, k *= 2) {
            unsigned int test = i * i;

            if (test > x && i == start + 1) {
                return start;
            }

            if (test > x) {
                return findNearest(i - k / 2, x);
            }
        }

        return 1;
    }
};
```

Figure 1: Finder class used in the complexity benchmark

The intended function is the integer square root:

$$\lfloor \sqrt{x} \rfloor ,$$

or, equivalently, the largest integer r such that

$$r^2 \leq x.$$

Remark. The implementation is not correct for every $x \geq 0$. For $x = 0$, the call `function(0)` enters `findNearest(0, 0)`. The loop tests `i` $= 0$, sees that `i*i > x` is false, exits the loop, and returns 1. The correct integer square root of 0 is 0.

There is also an implementation-level caveat in the line `unsigned int test = i * i;`. The product is computed in unsigned integer arithmetic, so it is not protected against wraparound if `i` becomes too

large for the chosen integer width. This is separate from the complexity question below, but a robust implementation would compute the square in a wider type.

The complexity question used the assumption $x > 0$ and assumed that the function returns $\lfloor \sqrt{x} \rfloor$. Thus the zero-input bug was not the point of the complexity stage. The point was whether the model could identify the variable that decreases in the recursion.

3 The Question

The final prompt was as follows:

> For inputs $x > 0$, assume that `Finder::function` returns $\lfloor \sqrt{x} \rfloor$.
>
> Estimate the time complexity of the implementation shown above as a function of the numeric input value x.
>
> Explain your reasoning carefully and state the final asymptotic complexity.

Full credit required the final answer

$$\Theta(\log^2 x)$$

and a derivation in terms of the distance from the current `start` value to $\lfloor \sqrt{x} \rfloor$.

4 Derivation

We will start with the loop. In one call to `findNearest`, the loop uses the variables `i` and `k`:

$$\texttt{i} = \texttt{start}, \qquad \texttt{k} = 1$$

at the first iteration. After every iteration, the update is

$$\texttt{i} \leftarrow \texttt{i} + \texttt{k}, \qquad \texttt{k} \leftarrow 2\texttt{k}.$$

Therefore the tested values of `i` are

$$\texttt{start}, \quad \texttt{start} + 1, \quad \texttt{start} + 3, \quad \texttt{start} + 7, \quad \ldots$$

After t loop iterations, the tested distance from `start` is

$$2^t - 1.$$

Now define

$$r = \left\lfloor \sqrt{x} \right\rfloor, \qquad d = r - \texttt{start}.$$

The variable d is the remaining distance to the integer square root. This is the quantity that changes in the recursive call. The numeric input x is passed unchanged.

The loop stops after it tests a value larger than r, because this is exactly when

$$\texttt{i}^2 > x.$$

It follows from the tested distances above that one loop level costs

$$\Theta(\log d)$$

for $d > 0$, up to the usual constant adjustment for very small d.

Suppose that the overshoot happens at iteration t. Then the current tested distance is too large:

$$2^t - 1 > d.$$

The previous tested distance was not too large:

$$2^{t-1} - 1 \leq d.$$

The recursive call starts from that previous tested value:

$$\texttt{start}' = \texttt{start} + 2^{t-1} - 1.$$

Thus the new distance is

$$\begin{aligned} d' &= r - \texttt{start}' \\ &= r - \left(\texttt{start} + 2^{t-1} - 1\right) \\ &= d - \left(2^{t-1} - 1\right). \end{aligned}$$

Since the overshoot was the first overshoot, $d < 2^t - 1$. Therefore

$$d' < (2^t - 1) - (2^{t-1} - 1) = 2^{t-1}.$$

At the same time $d \geq 2^{t-1} - 1$, so the new distance is at most about one half of the old one. Up to an additive constant, we can write the recursive cost as

$$T(d) = T(d/2) + \Theta(\log d). \tag{1}$$

Now we solve this recurrence. First, we can apply the Master theorem to the recurrence above. In the standard notation

$$T(n) = aT(n/b) + f(n),$$

we identify n with the distance d, and therefore

$$a = 1, \qquad b = 2, \qquad f(d) = \Theta(\log d).$$

Therefore

$$d^{\log_b a} = d^{\log_2 1} = d^0 = 1.$$

The non-recursive part can be written as

$$f(d) = \Theta(\log d) = \Theta\left(d^0 \log^1 d\right).$$

Thus this is the logarithmic case of the Master theorem, and it gives

$$T(d) = \Theta\left(d^0 \log^{1+1} d\right) = \Theta(\log^2 d).$$

It is useful to see the same result directly. Expanding the recurrence gives

$$T(d) = \Theta(\log d) + \Theta(\log(d/2)) + \Theta(\log(d/4)) + \cdots .$$

Let $m = \log_2 d$. Then this is the same as

$$\Theta(m) + \Theta(m-1) + \Theta(m-2) + \cdots + \Theta(1).$$

Therefore

$$T(d) = \Theta(m^2) = \Theta(\log^2 d).$$

For the original call, `start` $= 0$, and therefore

$$d = \left\lfloor \sqrt{x} \right\rfloor .$$

Substitution gives

$$\begin{aligned} T(x) &= \Theta(\log^2 \left\lfloor \sqrt{x} \right\rfloor) \\ &= \Theta\left(\left(\frac{1}{2}\log x\right)^2\right) \\ &= \Theta(\log^2 x). \end{aligned}$$

This is also the place where the most common wrong answer appears. It is tempting to write something like

$$T(x) = T(\sqrt{x}) + O(\log x).$$

But that is not the recurrence of this implementation. The recursive call does not receive $\sqrt{x}$. It receives the same x and a larger `start`. The hidden state variable is d, not x.

5 Model Results

The original final complexity stage was repeated three times for every model. Table 1 shows how many trials received full credit on the isolated complexity question.

Model	Full passes	Final complexity stage
`qwen3.6-27b-code`	0/3	missed
`qwen3.6-27b-mlx`	0/3	missed
`qwen3.6:35b`	0/3	missed
`qwen3.6-35b-a3b-code`	0/3	missed
`qwen3.6-35b-a3b-mlx`	0/3	missed
`qwen3-coder-next`	0/3	missed
`qwen3.5-122b`	3/3	$\Theta(\log^2 x)$
`gemma4-26b`	0/3	missed
`gemma4-31b`	0/3	missed
`gpt-oss-20b`	0/3	missed
`gpt-oss-120b`	0/3	missed

Table 1: Unhinted final complexity result. Full credit required the $\Theta(\log^2 x)$ answer and reasoning through the decreasing distance $d = \lfloor\sqrt{x}\rfloor - \texttt{start}$.

Only `qwen3.5-122b` solved the final complexity stage in all three unhinted trials. This does not mean that the other models could not solve the recurrence. It means that they usually did not introduce the right state variable on their own.

6 Hint Sensitivity

After the unhinted run, the final complexity question was rerun on the ten models that failed it. Each hint level used three trials per model, so each row in Table 2 summarizes thirty trials.

Hint level	What was added	Result
Weak	Warned that x does not change and asked the model to identify the shrinking quantity.	0/30 full passes. Models often noticed that x was unchanged, but still did not derive the distance recurrence.
Medium	Named $d = \lfloor\sqrt{x}\rfloor - \texttt{start}$ and asked for the recurrence in terms of d.	9/30 full passes and 3/30 partial passes. This was enough for `qwen3.6:35b`, `gemma4-26b`, and `gemma4-31b`.
Strong	Gave $T(d) = T(d/2)+O(\log d)$ and asked the model to solve it and substitute $d \le \sqrt{x}$.	27/30 full passes at the usual output budget. `gpt-oss-20b` also became 3/3 after increasing the output budget.

Table 2: Sensitivity of the complexity stage to added hints. The `qwen3.5-122b` model was not included because it already passed the unhinted final complexity stage.

The hint experiment separates two abilities. Once the recurrence

$$T(d) = T(d/2) + O(\log d)$$

was stated explicitly, almost all models completed the derivation. The hard part was earlier: reading the program as a recurrence in the hidden distance variable. For this benchmark, the main signal was not algebraic manipulation. The main signal was whether the model could choose the correct state variable before doing the algebra.

Conclusion

This benchmark used one small complexity question, but it exposed a useful difference between models. The code looks like a square-root search, but the recursive call does not reduce x. It only moves `start` closer to $\lfloor\sqrt{x}\rfloor$. Therefore the right analysis is

$$\begin{aligned} d &= \lfloor\sqrt{x}\rfloor - \texttt{start}, \\ T(d) &= T(d/2) + \Theta(\log d), \\ T(x) &= \Theta(\log^2 x). \end{aligned}$$

The models that missed the question usually missed this change of variable. That is exactly why this example is useful as a local benchmark: it tests whether the model can recover the algorithmic state from code, not only whether it can solve a recurrence after the recurrence has already been written down.