How to Reinvent the Bicycle

SERGEY BABKIN

Sergey Babkin has been employed as a Software Engineer for well over 20 years. His work experience includes SCO, Sybase, Microsoft, and, currently, Google. He likes to analyze and improve things. sab123@hotmail.com

Using programming puzzles as part of job applicant interviews has become common practice. While interviewing applicants, I've noticed two patterns in how they go about solving these puzzles. In this article, I examine these patterns and detail how programmers in general need to problem solve using the best of both patterns.

The Intuition and the System

In conducting recent job interviews, I've met a spate of junior engineer candidates with a similar issue: they quickly come up with a pretty good overall idea of the solution to a problem, and they can write code, but they fail to translate their solution into the code. They couldn't seem to organize the overall idea into components and then, step by step, work through the details and interdependencies of those components and sub-components, even with intense hinting from my side.

A bigger problem can always be seen as being composed of smaller, easier problems. The easier problems aren't necessarily easy, but two methods in dealing with them can be helpful: First, you can subdivide them further into even simpler problems. Second, as you try to solve a problem, you can gain an understanding of why it's difficult, and this often provides insight into solving the problem by avoiding it rather than overcoming it, by subdividing its parent problem differently. Not that all problems can be avoided: some things have to be overcome. The job applicants could come up with good ideas that solved difficult things that needed to be overcome, but they couldn't build a structure for the whole solution, where they could put these good ideas to good use.

To illustrate through an analogy, some time ago I read about an artist who would ask people to draw a bicycle from memory and then produce, as an art object, a bicycle based on the drawing. The results were art objects because they were completely non-functional. If I were to draw a bicycle without thinking, I would also produce something like that.

By spending some thought, any engineer should be able to reproduce a proper bicycle from the general logic: the function of the main components (wheels, steering, seat, pedals, chain) and the general considerations of the strength of the frame that shape it. The same logic can be used to check that none of the main components were forgotten: for example, if you forget about the chain, the pedals would be left disconnected from the rear wheel, so you'd have to remember it. Each component might be very non-trivial (the said chain took a long time to invent), but once you know the components, it should be impossible to put them in the wrong place.

This is something that should be done almost mechanically, with little mental effort. And yet these programming candidates could not do it. They tried to do it by intuition, but their intuition was not strong enough to handle a complex problem in one gulp, and they didn't know how to use the systematic approach either. The hints didn't help much; they didn't cause the right systematic associations.

Two Skills

There are really two orthogonal skills involved in solving these problems: to imagine the whole solution using highly developed intuition; to subdivide the problem and work through it iteratively, backtracking as necessary. Both are required to be a good engineer. A simple problem can be solved by using either of these skills alone. But even a moderately complex problem requires both skills; it's too big for intuition to figure out every detail, and too non-obvious for the systematic approach to find a good result in any reasonable time.

The problem I ask is actually quite difficult, too difficult for a perfectly adequate junior-to-mid-level engineer, and I'm not sure if I myself would have solved it well some 20 years ago. I know that I can solve it now, as it came from my real-life experience where I had to solve it really quickly from scratch. So I don't expect a good solution from this category of candidates; for them, a so-so solution is plenty good enough. Some of them actually do very well, producing a fully completed optimal solution.

There is a marked difference in how people with the one-sided development solve it, depending on which skill is their strong one. People with poor intuition and strong systematics produce a complete solution that is not very good. People with strong intuition and poor systematics get the right overall idea, figuring out the conceptual parts that I consider difficult and important (that the systematic group never figures out), only to fail miserably to work out all the details necessary to write the code. Not that the code doesn't get produced at all (though sometimes it doesn't), but what gets produced is closer to being an art object than working code.

Intuition, the Harder Skill

And that feels like a shame, because intuition is usually considered the harder skill to develop, requiring more time for development and being more rooted in natural ability. So there are people who could be good engineers if only they learned how to work systematically.

The trouble, I think, is that people are not really taught to do this kind of thinking in programming. Books and college courses describe the syntax of programming languages and the general picture but leave a void between these layers. People may learn this on their own from examples and practice. But the examples and practice tend to train the intuition, and people are left to figure out the systematic approach on their own, and they either figure it out or they don't. It looks like quite a few of the generally smart people either don't or take a long time to develop it. Yes, there are descriptions of how a problem has to be divided into the smaller parts, but they tend to miss the backtracking and the iterative redesign, making it look like intuition produces the right subdivision in one go. Not to say that there is anything wrong with intuition, it's my favorite thing, but the systematic approach allows you to stretch a good deal beyond the immediate reach of intuition, and to strengthen future intuition.

I've recently seen a question on Quora—"As you gain more experience, do you still write code that works but you don't know why?"—and this I think is exactly the difference between the intuitive and systematic solutions. Intuition might give you some code that works, or that possibly doesn't. The systematic approach lets you verify that what the intuition provided actually does what it's supposed to do and provides the stepping stones for the intuition to go further, both to fix what is going wrong and to produce more complex multi-leap designs.

Programming is not the only area with this kind of teaching problem. I think math has the same issue. The way proofs of various theorems are taught is usually not how the authors originally discovered them. These proofs get edited and adjusted a lot to make them look easier to understand. But then the teaching aspect of how to create new proofs through systematic trial and error gets lost.

Teaching the Two Skills

So how would you teach it? The bicycle example suggests that there is probably a general transferable skill too, and this skill can be trained by puzzle games like the classic "The Incredible Machine," where the goal is to build a Rube Goldberg contraption to accomplish the particular goal from a set of components. As in real life, the tasks there might include the extra components that look useful but don't really work out, or provide multiple ways to reach the goal. This of course requires that you achieve only one exact goal, while in programming you have to solve a whole class of related goals that include the corner cases. But this still might be a good place to start.

Perhaps the way to do it for programming is by walking through the solutions of complex problems, showing step by step how you can try the different approaches, follow through their elements, try to resolve the observed issues, and use this newly gained experience to find easier approaches. There are books built around somewhat different but closely related ideas: *Programming Pearls* and *More Programming Pearls* by Jon Bentley come to mind. *The Practice of Programming* by Brian Kernighan and Rob Pike, and, dare I say, my own *The Practice of Parallel Programming* are other examples.

A Systematic Puzzle

To give an example of what I think needs to be taught, I've decided to create a programming puzzle based on another, simpler interview problem that I used to use. The required insights in that problem are much smaller; it's much more about the systematic approach.

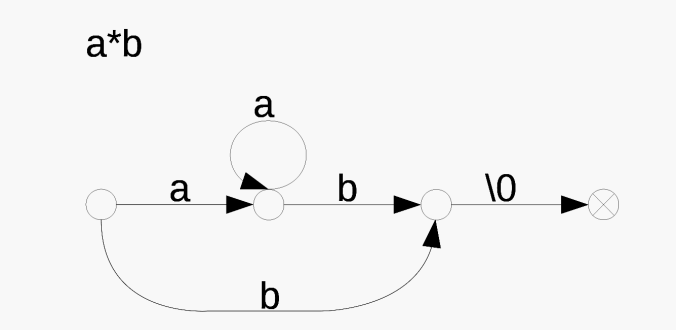

Figure 1: Finite state machine (FSM) for matching "a*b"

Since blindly remembering the solution to the problem is of no use to anyone, I want instead to show how better solutions can be born out of bad solutions. And it's not just brute force versus some ingenious algorithm. All the solutions to this problem are essentially brute force, but some of them are better and simpler than the others.

I'm going to start with the worst solution I can think of and then gradually show the better solutions. The puzzle for you, the reader, is to use the difficulties in these solutions as hints towards better solutions that would take you as far ahead as possible.

I wrote those solutions as I would do at an interview, without actually compiling and running the code on a computer, so they might contain bugs, but hopefully not many bad ones.

The problem is to write a matcher for the very simple regular expressions, that include only the operators "." (any character) and "*" (zero or more repetitions of the previous character). The "*" is greedy, consuming as many matching characters as possible. There is no escape character like backslash. The string must match completely, as if the regexp implicitly had anchors like "^" at the front and "$" at the end. And let's say that the string is in plain ASCII, so we don't need to bother with the wide characters.

The function declaration in plain C will be:

```
int match(const char *pattern, const char *text);
```

It will return 1 if the string matched the pattern and 0 if it didn't.

Let's start with the analysis. The first thing to notice about this problem is that some patterns in it are impossible to match. The "a*a" will never match anything because the greedy "a*" will consume all the "a"s, and the second "a" will never encounter a match. The same goes for ".*" followed by anything, because ".*" will consume everything to the end of the string.

The first solution proceeds in the most complicated way I can think of. You might have attended a college course on parsing that talked about the finite machine matcher for regular expressions. The most unsophisticated approach is to push this way blindly.

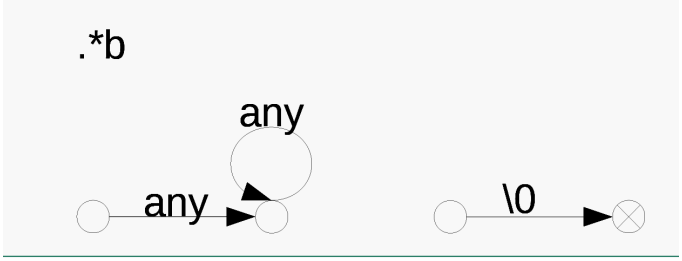

Figure 2: FSM for matching ".*b"

Before doing a finite machine, you'd really need to think of the state machine graphs you would be building for various regular expressions. I really could not get this code right until I had drawn the graphs.

Here are some examples: "a*b" is shown in Figure 1.

".*b" (with "any" meaning "everything but \0") is shown in Figure 2. This graph would never match anything, because it would never get into the final state (X). The FSM for "a*b*c" is shown in Figure 3, and "a*.*" in Figure 4.

Each state node of the finite machine graph would be represented by a dynamically allocated structure that has a plain array of the possible exits from that node, one per each character, and a flag showing that this node is final.

```
struct Node {

   Node *exits[256];

   int final;

};
```

The \0 could be handled as one of the normal exits, pointing to the final node. But there really isn't much point in having a separate node just to carry the final flag. It's easier to just set the final flag directly on a node that accepts an \0.

The graphs then become simpler, the graph in Figure 4 becoming as shown in Figure 5.

Since we're dynamically allocating the nodes, we need to take care of freeing them too. And that means taking care of keeping track of them while we use them. The inter-node links are no good for this purpose, since they branch multiple ways, and some graphs might even have some disconnected parts. But we can notice that there would always be as many nodes as elements (plain letters or starred letters) in the pattern, plus one. So we can just allocate the nodes as a single array and then free them as a single array.

This is a good time to stop and think about the question, is there really any point in bothering with the nodes? They will be strung generally sequentially, just like the original pattern. So why not just use the pattern directly? Indeed, this is a simpler approach. Time to change gears.

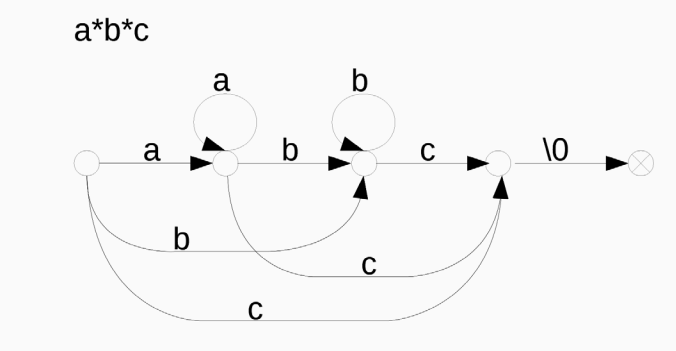

Figure 3: FSM for matching "a*b*c"

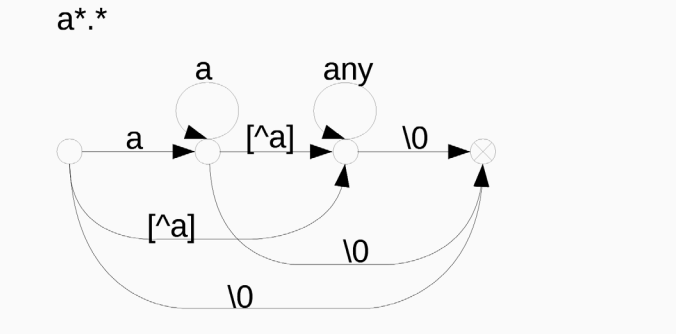

Figure 4: FSM for matching "a*.*"

Matching directly by pattern also has harder and easier versions. Again, let's start with the harder version.

The loop will be working in very much the same way as the matching loop in the parsed-pattern version (as some textbooks would teach) but will read the pattern directly from the string as it goes along.

Before writing the code, let's talk through the logic: as we read the next character of the text, we have a pointer to the next pattern element to parse. We parse the pattern element and match the text character to it. If the element is \0, we accept \0 and stop. If the element is starred and the character matches, we return the pattern back to the original position. If the element is starred and the character doesn't match, we try the next element from the pattern. If the element is '.', we accept everything but \0. If the element is another character, we accept it literally.

```
bool match(const char *pattern, const char *text) {
   char last_pattern = '\0';
   const char *p = pattern;
   for (const char *t = text; ; t++) {
      while (true) { // for starred sequences in pattern
         char element = *p++;
         if (element == '\0') {
            return *t == '\0';
         }

         if (*p == '*') {
            if (element == '.' && *t != '\0'
            || *t == element) { // matched
               --p; // return to the start of current element
               break;
            }

            // consume the star before reading the next element
            p++;
            continue;
         }
         if (element == '.' && *t == '\0'
         || *t != element) { // didn't match
            return false;
         }
         break;
      }
   }

   return false; // never reached
}
```

The inner loop is necessary to handle the sequences of multiple starred characters, such as "a*b*c" matching the "c". If we don't do the loop, "c" would get compared to "a", and the match will be considered failed.

The outer "for" loop here is interesting, without an exit condition. This is because the '\0' is matched inside the inner loop mostly in the same way as the normal characters: `(*t != element)` handles the unexpected '\0' in the same way as any other unexpected character. It's easy to start writing the loop with:

```
for (const char *t = text; *t != '\0'; t++)  {
   ...
}
return element == '\0';
```

But that would miss the situation where the pattern ends with a sequence of starred characters. This is something that is easy to miss, but it would be detected by a careful code analysis, a good unit test, or by a helpful interviewer. Then the code would need to be fixed by either bringing the handling of '\0' entirely into the inner loop as I have done here (there is no reason to be afraid of the loops that look nonstandard, they can be quite useful) or by moving the inner loop into a function and calling it again after the main loop (then the function would still have to handle '\0' as the next character of the text). The handling of '\0' in the inner loop is not that easy to get right; I got it working right with '.' only on the second attempt.

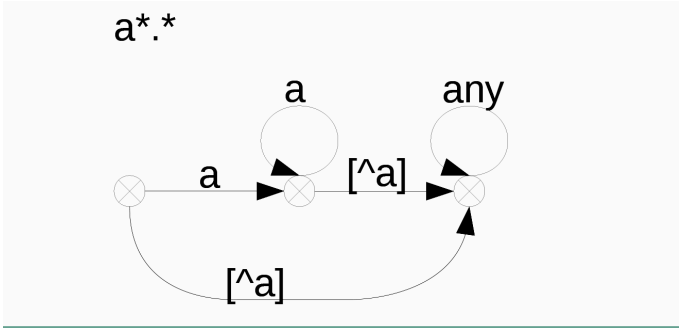

Figure 5: The improved FSM from Figure 4

The Value in Being Systematic

This is a good place to talk about how to fix a bug after it has been found. I've seen the people that are strong on intuition but not systematic start semi-randomly changing the spots that look vaguely plausible. I've literally seen a candidate do three wrong changes in a row, hoping every time that the issue will get resolved. This is the situation where thinking things through systematically really shines. Good questions to start with are, what do these values mean and how did their handling in the code diverge from this meaning? And then you can proceed to "Where did it happen?" and fix the bug. The same candidate, after I asked these questions, was able to find and resolve the bug on the first attempt in just a few seconds.

Returning to this solution, the problem that it solves is underspecified. It doesn't tell you what to do in case the pattern is invalid, either starting with a star or containing multiple stars in a row. This is by design, to see if the candidate will notice this and ask for a clarification, and my answer to this clarification question is, "What do you think is reasonable?" to see if the candidate is able to enumerate the pros and cons of different approaches: either return some error indication or handle it silently in some reasonable way.

I've made this solution do the silent handling, simply because it's easier to do in a small code snippet: it treats the "wrong" stars as literals. From the caller's standpoint it might be either good or bad: the good is that the caller won't have to handle the errors, and the bad is that the author of the incorrect pattern might be surprised by its effect and might never find out that it's incorrect.

But even this version is not great. The nested loops and re-parsing the pattern on each text character are convoluted; I got it right only on the second attempt. When the going gets hard, it's usually a good indication that a different approach should be tried.

What should the other approach be? It's up to your intuition to supply the ideas, for that's its line of work. This is why you need both intuition and systematics; one is not enough.

For this problem, it's much easier to go the other way around, iterating through the pattern and consuming the matching characters from the text:

```
bool match(const char *pattern, const char *text) {
   const char *t = text;
   for (const char *p = pattern; *p != 0; p++) {
      if (p[1] == '*') {
         if (*p == '.') {
            while (*t)
               ++t;
         } else {
            while (*t == *p)
               ++t;
         }
         ++p; // adjust to consume the star
      } else if (*p == '.') {
         if (*t++ == 0)
            return false;
      } else {
         if (*t++ != *p)
            return false;
      }
   }
   return *t == 0;
}
```

This version is much smaller and much easier to follow through. It explicitly selects by the type of each pattern element, so each one of them has its own code fragment, which avoids spreading its logic through the code and mixing it with the logic of the other elements. And all this makes the creation of bugs more difficult.

This whole problem is not very imaginative and can be solved well by just hammering out the code systematically. But this nice, short version contains an item that requires at least a little leap of intuition: it looks ahead by two characters, not just one, to detect whether the current pattern character is followed by a star. It's not something that's usually taught, but it makes the code a lot easier. As I like to say, it's not people for the programming patterns, it's programming patterns for the people. Don't be afraid to step away from a taught pattern if it makes your code better.

This version also has a theoretical foundation: it's a recursive-descent LL(1) parser of the text, except that the regular expressions define a non-recursive language, so there is no recursion. It really is perfectly per textbook; you've just got to pick the right textbook! It also parses, not a fixed grammar, but one given in the regular expression pattern. So it's an LL(2) parser of the pattern, with the nested LL(1) parsers of the matching substrings in the text. The 2 in LL(2) means that we're looking ahead by two characters. The pattern can also be parsed by an LL(1) parser, but looking ahead by two characters makes it easier.

Conclusion

This is the version that came to mind almost right away when I first thought about this problem. But I can't really say that it just popped into my mind out of nowhere. I do size up the different approaches in my mind intuitively and try the ones that look simpler first. It doesn't mean that this first estimation is always right. Sometimes I go pretty deep with one approach before deciding to abandon it and apply the lessons learned to another approach. And sometimes this other approach ends up being even worse, but the lessons learned there help to get through the logjam of the first approach.

So if you start with poor approaches, you can still arrive at better ones by listening to the hints that the code gives to you as you write it. When you see an easier way to go, use it. You can also power through the difficult approaches systematically to the successful end, but that tends to be much more difficult than switching the approach to an easier one. Intuition and systematic logic working hand-in-hand can get you much farther than either one of them alone.