Overview of String Matching Algorithms

This lesson will provide an overview of matching algorithms.

WE'LL COVER THE FOLLOWING

String Matching

String-matching consists of finding one or all of the occurrences of a string ("pattern") in a text. The strings are built over a finite set of characters, called "alphabet".

There are lots of algorithms that solve this problem; here's a short list from Wikipedia:

Algorithm	Preprocessing	Matching	Space
Naive string-search	none	`O(nm)`	none
Rabin–Karp	`O(m)`	average `O(n + m)`, worst `O((n-m)m)`	`O(1)`
Knuth–Morris–Pratt	`O(m)`	`O(n)`	`O(m)`

Boyer–Moore	`O(m + k)`	best `O(n/m)`, worst `O(mn)`	`O(k)`
Boyer–Moore-Horspool	`O(m + k)`	best `O(n/m)`, worst `O(mn)`	`O(k)`

`m` - the length of the pattern `n` - the length of the text `k` - the size of the alphabet

Naive algorithm

The naive algorithm tries to match the pattern at each position of the text:

```
Pattern = Hello
Text = SuperHelloWorld

   SuperHelloWorld
1. Hello <- XX
2.  Hello <- XX
3.   Hello <- XX
4.    Hello <- XX
5.     Hello <- XX
6.      Hello <- OK!
```

In the example above we're looking for "Hello" in "SuperHelloWorld". As you can see, the naive version tries each position until it finds the "Hello" at the 6th iteration.

The main difference between the naive way and the other algorithms is that the faster algorithms use additional knowledge about the input pattern. That way, they can skip a lot of fruitless comparisons.

To gain that knowledge, they usually build some lookup tables for the pattern in the preprocessing phase. The size of lookup tables is often tied to the size of the pattern and the alphabet.

Boyer Moore algorithm

In the above case we can skip most of the iterations, as we can observe that when we try to match `Hello` in the first position, there's a difference at the last letter `o` vs `r`. In fact, since `r` doesn't occur in our pattern at all, we can actually move 5 steps further.

actually move 3 steps further.

```
Pattern = Hello
Text = SuperHelloWorld

   SuperHelloWorld
1. Hello <- XX,
2.      Hello <- OK
```

We have a match with just 2 iterations!

> The rule that was used in that example comes from the Boyer Moore algorithm - it's called The Bad Character Rule.

Before C++17

In C++ string matching is implemented through `std::search`, which finds a range (the pattern) inside another range:

```
template< class ForwardIt1, class ForwardIt2 >
ForwardIt1 search( ForwardIt1 first, ForwardIt1 last,
                   ForwardIt2 s_first, ForwardIt2 s_last );
```

Before C++17, you had no control over the algorithm selection inside `std::search`. The complexity of `std::search` was specified as `O(mn)` - so it was usually the naive approach. Now, in C++17, you have a few more options.

To be precise, there's also `string::find` that works exclusively with character sequences. There might be different implementations of this method, and you don't have control over the algorithm that is used inside.

In the examples section, you'll see some performance experiments that also compare the new algorithms for `std::search` with `string::find`.

Now we know that string matching is implemented via std::search in C++. Let's discuss how C++17 has updated std::search and some new searcher algorithms available in C++ 17.