Knuth-Morris-Pratt

Martin

What is KMP?

- Dynamic programming?
- Recursion?
- The hardest of the 4 common problem solving paradigms?
- The easiest of the 4 common problem solving paradigms?

Problem

Ctrl + f

Given a text and a pattern, return all occurrences of the pattern in the text.

Example

Let the pattern P =

a	b	a	b

Let the text T =

b	a	c	b	a	b	a	b	a	b	b	a	b	a	b

Naive algorithm?

Just “slide” the pattern across the text on character at a time

O((n-m+1)m)

Example

How can we improve this?

Use KMP!

Very similar to the naive, but we can make bigger steps if some conditions are met...

If the start of the pattern occurs again later in the pattern, we can use this to skip some steps

Example

P =

a	b	a	b

T =

b	a	c	b	a	b	a	b	a	a	b	c	b	a	b

Example

P =

a	b	a	b

T =

b	a	c	b	a	b	a	b	a	a	b	c	b	a	b

We see the first character matches and the second doesn't.

b != c, but we also know in our pattern that a != b...

Example

P =

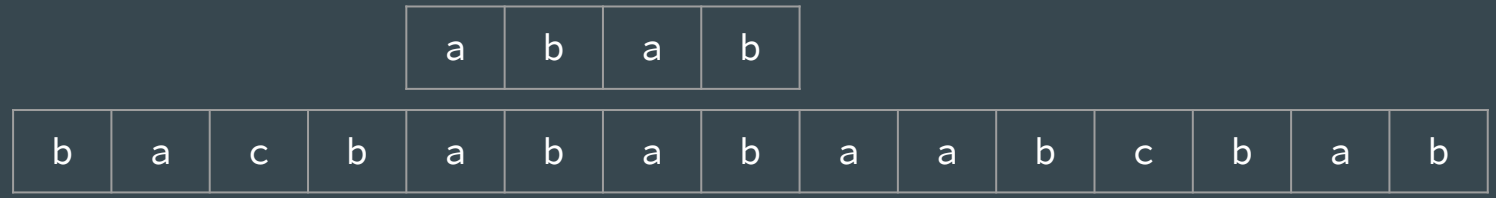
a	b	a	b

T =

b	a	c	b	a	b	a	b	a	a	b	c	b	a	b

Since the first and second characters in P are different, we know we can slide over 2 spaces!

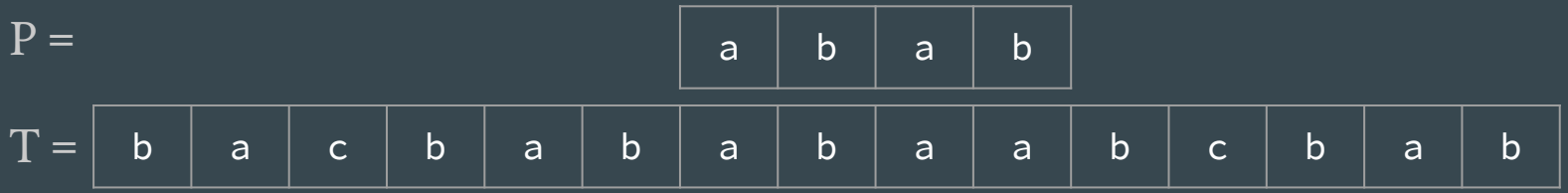
Example



One full match found!

We also see that we can slide over 2 just like before.

Example



Example

P =

a	b	a	b

T =

b	a	c	b	a	b	a	b	a	a	b	c	b	a	b

This time we slide ahead 3 spaces.

More partial matches.

Example

P =

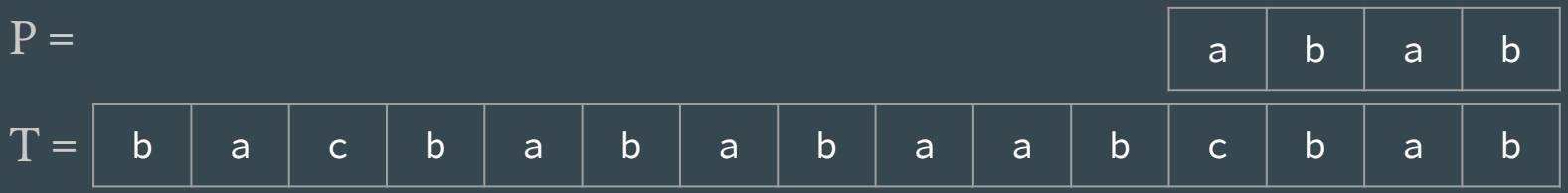
											a	b	a	b

T =

b	a	c	b	a	b	a	b	a	a	b	c	b	a	b

Slide 2 again and we've hit the end.

Example



We are finished searching the string.

7 comparisons vs. 12 comparisons

Almost a 50% speedup on smaller strings!*

The Backtable

How did the algorithm know how far to skip ahead?

We preprocess the pattern to build a “backtable” which tells us.

a	b	a	b

-1	0	0	1	2

The Backtable

To build this table we find the longest proper prefix of `pattern[0..i]` that is also a suffix of `pattern[0..i]` for each `i`.

In this pattern we see that the substring aba has common prefix a.

ab ab ab(a)ba ababaa ababaaa ababaaab

a	b	a	b	a	a	a	b

-1	0	0	1	2	3	1	1	2

The Algorithm

```
vector<int> build_backtable(string pattern) {
    vector<int> backtable = vector(pattern.size()+1);
    backtable[0] = -1;
    for (int i = 1; i < pattern.size()); i++) {
        int pos = backtable[i-1];
        while (pos != 1 && pattern[i-1] != pattern[pos])
            pos = backtable[pos];
        backtable[i] = pos + 1;
    }

    return backtable;
}
```

The Algorithm

```
vector<int> kmp(string text, string pattern) {
    vector<int> matches;
    vector<int> backtable = build_backtable(pattern);
    i, j = 0;
    while i < text.size() {
        while (j != 1 && (j == pattern.size() || text[i] != pattern[j]))
            j = backtable[j];
        i++; j++;
        if (i == j)
            matches.push_back(i);
    }
}
```

A few notes

- KMP is not limited to only strings! Any indexable structure will work.
- GNU C `strstr()` and Python `indexOf()` use KMP but C++ `string.find()` and Java `String.index()` do not.
  - If you want to KMP on arrays (eg ints) you'll have to implement it yourself.
- There are many string searching algorithms with different use cases.
  - Naive: 0 preprocessing time | $O((n-m+1)m)$ matching time
  - Knuth-Morris-Pratt: $\Theta(m)$ preprocessing time | $\Theta(n)$ matching time
  - Rabin-Karp: $\Theta(m)$ preprocessing time | $O((n-m+1)m)$ matching time
  - Finite Automaton: $O(m|\Sigma|)$ preprocessing time | $\Theta(n)$ matching time

That’s it! Thanks!