KMP String Searching

Bruce Merry

IOI Training Mar 2014

Outline

The Problem

- Given a string H (haystack) and another string N (needle), find all places where N occurs in H.

The Problem

- Given a string H (haystack) and another string N (needle), find all places where N occurs in H.
- These places might overlap.

The Problem

- Given a string H (haystack) and another string N (needle), find all places where N occurs in H.
- These places might overlap.
- The “strings” might not be made up of English letters, but of numbers or other objects.

Simple Solution

- Try every possible substring to H and compare to N

Simple Solution

- Try every possible substring to H and compare to N
- Implemented by `std::search`

Simple Solution

- Try every possible substring to H and compare to N
- Implemented by `std::search`
- Complexity is $O(|H| \cdot |N|)$.

Some String Algorithms

- Knuth-Morris-Pratt: Runs in $\Theta(|H| + |N|)$
- Boyer-Moore: Worst-case $O(|H| + |N|)$, much better for normal text
- Horspool: Simplified Boyer-Moore, worst case $O(|H| \cdot |N|)$
- Rabin-Karp: $O(|H| + |N| + m|N|)$ for m matches, except for pathological cases
- Aho-Corasick: Generalized KMP that searches for multiple strings

Some String Algorithms

- Knuth-Morris-Pratt: Runs in $\Theta(|H| + |N|)$
- Boyer-Moore: Worst-case $O(|H| + |N|)$, much better for normal text
- Horspool: Simplified Boyer-Moore, worst case $O(|H| \cdot |N|)$
- Rabin-Karp: $O(|H| + |N| + m|N|)$ for m matches, except for pathological cases
- Aho-Corasick: Generalized KMP that searches for multiple strings

For contests, Boyer-Moore and Horspool are not useful.

Overview

- Process one character from *H* at a time, keeping track of the longest prefix of *N* matching at this point.
- If the next letter of *H* doesn't match our current prefix of *N*, we fall back to a shorter prefix and try again.

A B C A B A B A C A B A B A C A B A D

A B A C A B A D

Overview

- Process one character from *H* at a time, keeping track of the longest prefix of *N* matching at this point.
- If the next letter of *H* doesn't match our current prefix of *N*, we fall back to a shorter prefix and try again.

A B C A B A B A C A B A B A C A B A D
A B A C A B A D

Overview

- Process one character from *H* at a time, keeping track of the longest prefix of *N* matching at this point.
- If the next letter of *H* doesn't match our current prefix of *N*, we fall back to a shorter prefix and try again.

A B C A B A B A C A B A B A C A B A D

A B A C A B A D

Overview

- Process one character from *H* at a time, keeping track of the longest prefix of *N* matching at this point.
- If the next letter of *H* doesn't match our current prefix of *N*, we fall back to a shorter prefix and try again.

A B C A B A B A C A B A B A C A B A D
A B A C A B A D

Overview

- Process one character from *H* at a time, keeping track of the longest prefix of *N* matching at this point.
- If the next letter of *H* doesn't match our current prefix of *N*, we fall back to a shorter prefix and try again.

A B C A B A B A C A B A B A C A B A D

A B A C A B A D

Overview

- Process one character from *H* at a time, keeping track of the longest prefix of *N* matching at this point.
- If the next letter of *H* doesn't match our current prefix of *N*, we fall back to a shorter prefix and try again.

A B C A B A B A C A B A B A C A B A D

A B A C A B A D

Overview

- Process one character from *H* at a time, keeping track of the longest prefix of *N* matching at this point.
- If the next letter of *H* doesn't match our current prefix of *N*, we fall back to a shorter prefix and try again.

A B C A B A B A C A B A B A C A B A D

A B A C A B A D

Overview

- Process one character from *H* at a time, keeping track of the longest prefix of *N* matching at this point.
- If the next letter of *H* doesn't match our current prefix of *N*, we fall back to a shorter prefix and try again.

A B C A B A B A C A B A B A C A B A D

A B A C A B A D

Overview

- Process one character from *H* at a time, keeping track of the longest prefix of *N* matching at this point.
- If the next letter of *H* doesn't match our current prefix of *N*, we fall back to a shorter prefix and try again.

A B C A B A B A C A B A B A C A B A D

A B A C A B A D

Overview

- Process one character from *H* at a time, keeping track of the longest prefix of *N* matching at this point.
- If the next letter of *H* doesn't match our current prefix of *N*, we fall back to a shorter prefix and try again.

A B C A B A B A C A B A B A C A B A D

A B A C A B A D

Overview

- Process one character from *H* at a time, keeping track of the longest prefix of *N* matching at this point.
- If the next letter of *H* doesn't match our current prefix of *N*, we fall back to a shorter prefix and try again.

A B C A B A B A C A B A B A C A B A D

A B A C A B A D

Overview

- Process one character from *H* at a time, keeping track of the longest prefix of *N* matching at this point.
- If the next letter of *H* doesn't match our current prefix of *N*, we fall back to a shorter prefix and try again.

A B C A B A B A C A B A B A C A B A D

A B A C A B A D

Overview

- Process one character from *H* at a time, keeping track of the longest prefix of *N* matching at this point.
- If the next letter of *H* doesn't match our current prefix of *N*, we fall back to a shorter prefix and try again.

A B C A B A B A C A B A B A C A B A D

A B A C A B A D

Overview

- Process one character from *H* at a time, keeping track of the longest prefix of *N* matching at this point.
- If the next letter of *H* doesn't match our current prefix of *N*, we fall back to a shorter prefix and try again.

A B C A B A B A C A B A B A C A B A D

A B A C A B A D

Overview

- Process one character from *H* at a time, keeping track of the longest prefix of *N* matching at this point.
- If the next letter of *H* doesn't match our current prefix of *N*, we fall back to a shorter prefix and try again.

A B C A B A B A C A B A B A C A B A D
A B A C A B A D

Overview

- Process one character from *H* at a time, keeping track of the longest prefix of *N* matching at this point.
- If the next letter of *H* doesn't match our current prefix of *N*, we fall back to a shorter prefix and try again.

A B C A B A B A C A B A B A C A B A D

A B A C A B A D

Overview

- Process one character from *H* at a time, keeping track of the longest prefix of *N* matching at this point.
- If the next letter of *H* doesn't match our current prefix of *N*, we fall back to a shorter prefix and try again.

A B C A B A B A C A B A B A C A B A D

A B A C A B A D

Overview

- Process one character from *H* at a time, keeping track of the longest prefix of *N* matching at this point.
- If the next letter of *H* doesn't match our current prefix of *N*, we fall back to a shorter prefix and try again.

A B C A B A B A C A B A B A C A B A D

A B A C A B A D

Overview

- Process one character from *H* at a time, keeping track of the longest prefix of *N* matching at this point.
- If the next letter of *H* doesn't match our current prefix of *N*, we fall back to a shorter prefix and try again.

A B C A B A B A C A B A B A C A B A D

A B A C A B A D

Overview

- Process one character from *H* at a time, keeping track of the longest prefix of *N* matching at this point.
- If the next letter of *H* doesn't match our current prefix of *N*, we fall back to a shorter prefix and try again.

A B C A B A B A C A B A B A C A B A D

A B A C A B A D

Overview

- Process one character from *H* at a time, keeping track of the longest prefix of *N* matching at this point.
- If the next letter of *H* doesn't match our current prefix of *N*, we fall back to a shorter prefix and try again.

A B C A B A B A C A B A B A C A B A D

A B A C A B A D

Overview

- Process one character from *H* at a time, keeping track of the longest prefix of *N* matching at this point.
- If the next letter of *H* doesn't match our current prefix of *N*, we fall back to a shorter prefix and try again.

A B C A B A B A C A B A B A C A B A D

A B A C A B A D

Overview

- Process one character from *H* at a time, keeping track of the longest prefix of *N* matching at this point.
- If the next letter of *H* doesn't match our current prefix of *N*, we fall back to a shorter prefix and try again.

A B C A B A B A C A B A B A C A B A D

A B A C A B A D

Overview

- Process one character from *H* at a time, keeping track of the longest prefix of *N* matching at this point.
- If the next letter of *H* doesn't match our current prefix of *N*, we fall back to a shorter prefix and try again.

A B C A B A B A C A B A B A C A B A D

A B A C A B A D

Overview

- Process one character from *H* at a time, keeping track of the longest prefix of *N* matching at this point.
- If the next letter of *H* doesn't match our current prefix of *N*, we fall back to a shorter prefix and try again.

A B C A B A B A C A B A B A C A B A D

A B A C A B A D

Failure Function

Given a prefix `N[:i]`, what is the largest $j < i$ such that `N[:j]` is a suffix of `N[:i]`?

- Pre-computed, stored in a table `f[i]`

Failure Function

Given a prefix `N[:i]`, what is the largest $j < i$ such that `N[:j]` is a suffix of `N[:i]`?

- Pre-computed, stored in a table `f[i]`
- Useful to set `f[0] = -1`

Failure Function

Given a prefix `N[:i]`, what is the largest $j < i$ such that `N[:j]` is a suffix of `N[:i]`?

- Pre-computed, stored in a table `f[i]`
- Useful to set `f[0] = -1`
- `f[i]`, `f[f[i]]`, `f[f[f[i]]]` etc. give all the prefixes of `N[:i]` that are also suffixes.

Failure Function

Given a prefix `N[:i]`, what is the largest $j < i$ such that `N[:j]` is a suffix of `N[:i]`?

- Pre-computed, stored in a table `f[i]`
- Useful to set `f[0] = -1`
- `f[i]`, `f[f[i]]`, `f[f[f[i]]]` etc. give all the prefixes of `N[:i]` that are also suffixes.
- Need to compute in linear time

Failure Function Computation

It can be computed by dynamic programming:

- Set `f[0] = -1`

Failure Function Computation

It can be computed by dynamic programming:

- Set `f[0] = -1`
- If `N[:i].endswith(N[:j])`, then `N[:i-1].endswith(N[:j-1])`. Thus j is $f^r(i-1)+1$ for some repeat count r.

Failure Function Computation

It can be computed by dynamic programming:

- Set `f[0] = -1`
- If `N[:i].endswith(N[:j])`, then `N[:i-1].endswith(N[:j-1])`. Thus j is $f^r(i-1)+1$ for some repeat count r.
- Only need to check that `N[i-1] == N[j-1]`

Failure Function Computation

It can be computed by dynamic programming:

- Set `f[0] = -1`
- If `N[:i].endswith(N[:j])`, then `N[:i-1].endswith(N[:j-1])`. Thus j is $f^r(i-1)+1$ for some repeat count r.
- Only need to check that `N[i-1] == N[j-1]`
- Just try all values of j until one fits or $j = -1$.

Failure Function Computation

It can be computed by dynamic programming:

- Set `f[0] = -1`
- If `N[:i].endswith(N[:j])`, then `N[:i-1].endswith(N[:j-1])`. Thus j is $f^r(i-1)+1$ for some repeat count r.
- Only need to check that `N[i-1] == N[j-1]`
- Just try all values of j until one fits or $j = -1$.

Exercise: prove that this takes only linear time.

Failure Function Code


```
int L = N.size();
vector<int> fail(L + 1);
fail[0] = -1;
for (int i = 1; i <= L; i++)
{
    int f = fail[i - 1];
    while (f >= 0 && N[f] != N[i - 1])
        f = fail[f];
    fail[i] = f + 1;
}
```

Knuth-Morris-Pratt Code


```
int match = 0;
for (char c : H)
{
    while (match >= 0 && N[match] != c)
        match = fail[match];
    match++;
    if (match == int(N.size()))
        cout << "Found!\n";
}
```

Summary

- Always $O(|H| + |N|)$ (no pathological cases)

Summary

- Always $O(|H| + |N|)$ (no pathological cases)
- Works with arbitrarily-large alphabet

Summary

- Always $O(|H| + |N|)$ (no pathological cases)
- Works with arbitrarily-large alphabet
- Simple to implement

Summary

- Always $O(|H| + |N|)$ (no pathological cases)
- Works with arbitrarily-large alphabet
- Simple to implement
- Requires $O(N)$ memory

Summary

- Always $O(|H| + |N|)$ (no pathological cases)
- Works with arbitrarily-large alphabet
- Simple to implement
- Requires $O(N)$ memory
- Can stream in H