Plan:
Knuth-Morris-Pratt
Aho-Corasick
Rabin-Karp

Algorithms on Strings
Pattern Matching
Wild-card Matching
Compute a distance between two strings
Compute a longest substring
Compute a cheapest tree connecting all
given strings (phylogeny tree)
Compute a shortest superstring of all
strings

Pattern Matching
Let $T$ be a string of length $N$ over a finite alphabet $\Sigma$ and $P$ be a string of length $M$
over $\Sigma$.

In a pattern matching problem we search
for all occurrences of a pattern $P$ in a
text $T$.

Brute-Force Algorithm
It runs in time $O(nm)$
Example of worst case:
$T = \text{aaa} \ldots \text{ah}$
$P = \text{aaah}$
may occur in images and DNA
sequences
unlikely in English text

Deterministic Finite Automaton
A finite automaton $M$ is defined as a 5-tuple
$M = (\Sigma, Q, q_0, A, \delta)$
$\Sigma$ is the alphabet
$Q$ is the set of states
$q_0 \in Q$ is the start state
$A \subseteq Q$ is the set of accept states
$\delta : Q \times \Sigma \rightarrow Q$ is the transition function
$L(M) =$ the language of machine $M$
set of all strings machine $M$ accepts
**The Knuth-Morris-Pratt Algorithm (1976)**

Build DFA from pattern
Run DFA on multiple texts

- The alphabet is \{a, b\}.
- The pattern is \texttt{aa b a a a b b}.

To create a DFA we consider all prefixes \texttt{\varepsilon}, \texttt{a}, \texttt{aa}, \texttt{aab}, \texttt{aaba}, \texttt{aabaa}, \texttt{aabaaab}, \texttt{aabaaabb}.

These prefixes are states. The initial state is \texttt{\varepsilon} (empty string). The pattern is the accept state.

---

**DFA Construction \texttt{a a b a a a b b}**

Matched: \texttt{a a b a a a b b}

---

**The Knuth-Morris-Pratt Algorithm (1976)**

1970 Cook published a paper about a possibility of existence of such algorithm.

Knuth and Pratt developed an algorithm.

Morris discovered the same algorithm.
Building a DFA

What is the worst-case runtime of building a DFA?

\[ O(M^3 \Sigma) \]

\[ M = \text{pattern.length();} \]

\[ \Sigma = \text{alphabet.size();} \]

KMP eliminates the need to compute the entire transition function.

The KMP Algorithm - Motivation

Algorithm compares the pattern to the text in left-to-right, but shifts the pattern more intelligently than the brute-force algorithm.

When a mismatch occurs, what is the most we can shift the pattern so as to avoid redundant comparisons?

Building a DFA

KMP

How much can a string overlap with itself at each position?

Compute the length of the longest prefix of \( P \) that is a proper suffix of \( P \).

It determines where to go whenever there is a mismatch in the next letter.

Matching

\[
\begin{array}{cccc}
\text{a} & \text{b} & \text{a} & \text{b} \\
0 & 0 & 1 & 2 \\
\end{array}
\]

KMP

\[
\begin{array}{cccc}
\text{a} & \text{b} & \text{a} & \text{b} \\
0 & 0 & 1 & 2 \\
\end{array}
\]
It determines where to go whenever there is a mismatch in the next letter.
Failure Function

Consider all prefixes \( w[\cdot] \) of a pattern, define 
\[ \pi[k] = \max\{j < k \mid w[j] \text{ is a suffix } w[k]\} \]

\( \pi[k] \) is called a failure function, since it represents only backward transitions, in other words, it determines where to go whenever there is a mismatch in the next letter.

\( "aabaab", \pi = \{0, 1, 0, 1, 2, 3\} \)
\( "aabaab", \pi = \{0, 1, 0, 1, 2, 2, 3\} \)
\( "aaabaabaaa", \pi = \{0,1,2,0,1,2,0,1,2,3\} \)

```
Failure Function
int[] pi = new int[pattern.length()];
int x = 0;
for(int p = 1; p < pattern.length(); p++)
{
    while(x > 0 &&
        pattern.charAt(x) != pattern.charAt(p))
        x = pi[x-1];
    if(pattern.charAt(x) == pattern.charAt(p)) x++;
    pi[p] = x;
}
```

Matching

```
int x = 0;
for(int k = 0; k < text.length(); k++)
{
    while(x > 0 &&
        pattern.charAt(x) != text.charAt(k))
        x = pi[x-1];
    if(pattern.charAt(x) == text.charAt(k)) x++;
    if(x == pattern.length()) return true;
}
```

Applications

**DNA matching:**
DNA consists of small molecules called nucleotides. There are four of them Adenine, Cytosine, Guanine and Thymine. Therefore, \( \{A, C, G, T\} \) creates an alphabet.

**Protein matching:**
Proteins are composed of amino acids. There are basically 20 amino acids. Hence, a protein can be represented as a string over 20 letters.
The Aho-Corasick Algorithm (1986)

The algorithm preprocesses the set of patterns.

Patterns \{he, she, his, hers\}

The Aho-Corasick Algorithm

We still use the longest suffix rule. If we fail on making a transition from a node N to its child, we transition to a node M, where the string that defines M is the farthest node (longest prefix) from the root which is also a suffix of the string we had matched when we failed (removing the first transition).

The only difference is that instead of traversing a single string left-to-right we now have to traverse a trie.

The Rabin-Karp Algorithm (1981)

The algorithm uses the idea of hashing

The main idea

\begin{align*}
\text{pattern} &= 4848 \\
\text{text} &= 16180339887498948482045
\end{align*}

We do not match a string against a given pattern, but rather compare their hash codes.

The main idea

\begin{align*}
\text{pattern} &= 4848 \mod 71 = 20 \\
16180339887498948482045 &\mod 71 = 56 \\
1618 &\mod 71 = 56 \\
6180 &\mod 71 = 3 \\
1803 &\mod 71 = 28
\end{align*}

We read the text in the number of characters equal to the length of the pattern, compute its hash code and compare with the pattern hash code.
What is its complexity?

\[ M = \text{pattern.length()} \]
\[ N = \text{text.length()} \]

Similar to a brute-force matching…

The key idea of improving the algorithm is in computing a hash code in \(O(1)\).

**Computing a hash code**

How can we get from 145 to 456?

We will do this by creating a chain of operations

\[ 145 \rightarrow 45 \rightarrow 450 \rightarrow 456 \]

Remove the leading digit, multiply by a base, add a single digit. It takes \(O(1)\) to compute a hash code from the previous value.

**Example**

*Given:* a hash code for 31729

\[ 31729 \mod 41 = 36 \]

*Task:* compute a hash code for 17295.

Observe,

\[ 17295 = (31729 - 3 \cdot 10^4) \cdot 10 + 5 \]

Example

*Given:* a hash code for 31729

\[ 31729 \mod 41 = 36 \]

*Task:* compute a hash code for 17295.

Rabin-Karp formalized

Let \(P[1 \ldots m]\) be a pattern and \(T[1 \ldots n]\) be a text. We define a pattern

\[ P = 10^{m-1}P[1] + 10P[m-1] + \ldots + P[m] \]

and a shift in the text:

\[ t_s = 10^{m-1}T[s+1] + 10T[s+m-1] + \ldots + T[s+m] \]

The value \(t_{s+1}\) can be obtained from \(t_s\) by

\[ t_{s+1} = (t_s - 10^{m-1}T[s+1]) \cdot 10 + T[s+m+1] \]
We said “31729%41 is already computed”
How would you compute it fast?

Horner’s Rule
\[ a x^4 + b x^3 + c x^2 + d x + e = e + x (d + x (c + x (b + a x)) \]

Implementation
```java
public int search(String T, String P){
    int M = P.length(), N = T.length();
    int dM = 1, h1 = 0, h2 = 0;
    int p = 3355439; /*pick it at random */
    int d = 256;        /* radix */
    for(int j = 1; j < M; j++)
        dM = (d*dM) % p;
    for(int j = 0; j < M; j++)
        h1 = (h1 *d + P.charAt(j)) % p;
    h2 = (h2*d + T.charAt(j)) % p;
    if(h1 == h2) return 0;
    for(int i = M; i < N; i++)
        h2 = (h2 - T.charAt(i - M) * dM) % p;
        h2 = (h2*d + T.charAt(i)) % p;
        if(h1 == h2) return i - M + 1;
    return -1;
}
```

Implementation (cont.)

False match
T == P mod q
What do we do in a case of false match?
When we found a match we can check the match by char comparison.

False match
Theorem.
There’s a pattern of length M and a text of length N. Pick a random prime \( \epsilon [2,..., M N^2] \).
The probability of getting a false match anywhere in the string is at most 2.53/N.