String Matching
Chapter 34
(Skip Rabin-Karp for now, may return to it later!)

The string matching problem is to find if a pattern P[1..m] occurs within text T[1..n]. We have already examined the example of approximate string matching using dynamic programming. In this lecture we will examine efficient ways to perform exact string matching. Note that string matching is useful in more cases than just searching for words in text. String matching also applies to other problems, for example, matching DNA patterns in the human genome.

Naïve String Matching Algorithm

Find all valid shifts of P using a loop:

```
Naïve-Match(T,P)
    n← length[T]
    m← length[P]
    for s← 0 to n-m
        do if P[1..m]=T[s+1,s+m]  then P matches T at index s+1
```

Example:

```
T    =    ILOVEALGORITHMS
P    =         ALGOR
```

O(n+m) in this case, not much duplication of P in T

```
T    =    AAAAAAAAAAAAAA
P    =    AAAAAB
           AAAAAB
               etc.
```

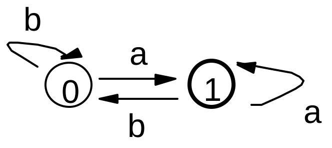
O(mn) in this case, for each of the n characters we have to go through all m chars of P

The naïve string matcher is slow in some cases, although in many cases it actually works pretty good and should not be ignored, especially since it is easy to implement. For small n or cases when the text and pattern differ, this is one of the best methods to use.

String-Matching Automata

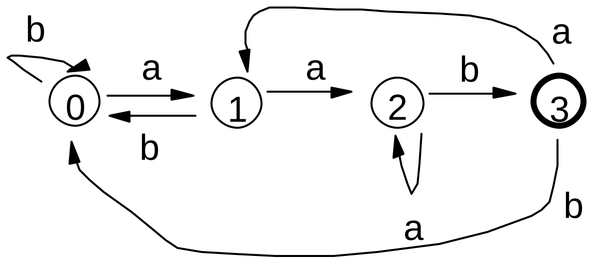
One way to improve string matching runtime is to construct a finite automaton that has knowledge about what we should compare next. These run in O(n) time but may require a large amount of time to construct the automaton.

Recall that a finite automaton is just a set of states with transitions between those states. For example, the automaton below accepts any string ending with an "a". bbba is accepted, as is bbaba. However, string bbbab would be rejected since it does not end in a final state.



The idea is to build an automaton that can take the input string, and consider all input possibilities to determine whether or not the pattern has been found.

As a more complex example, consider the automaton below that recognizes the string "aab" when the alphabet contains only a and b.



Example: abaabbaaaaabaab results in 3 matches. This automaton recognizes the pattern "aab".

We start in state 0, and feed it input characters from the text. If we ever end up in state 3, then we have found a match. If we get an aab, then we end up in state three. However, if we have aaa, even though there is no match, we should stay in state 2, since the previous two characters were a, and if the next one is b then we do have a match. Consequently, the automaton considers all possible inputs we may get, and moves to the correct state that is necessary to determine a match.

Given such an automaton, the algorithm to match the automaton is simple:

```
Finite-Automata-String-Matcher(T, $d$ ,m)          ; Text, transition function, m=final state
        n $\leftarrow$ length[T]
        q $\leftarrow$ 0
        for i $\leftarrow$ 1 to n  do
            q $\leftarrow$ $d$ (q,T[i])
            if q=m then s $\leftarrow$ i-m          ; Pattern at location s
                                                    ; final state is the 'mth' character in P
```

Hard part: Computing the automaton. We will only give a high level description of the algorithm here

```
    Compute-Transition-Function(P, $\Sigma$ )
        m $\leftarrow$ length[P]
        for q $\leftarrow$ 0 to m
            do for each character a in $\Sigma$
                k $\leftarrow$ state where a$P_q$ is a suffix of P
                $d$ (q,a) $\leftarrow$ k
```

Construction of the automaton depends upon the size of the alphabet, and can be constructed in time O(m|Σ|), where Σ is the alphabet. Works good for a small alphabet size.

Knuth-Morris-Pratt Algorithm

This is a famous linear-time running string matching algorithm that achieves a Θ (m+n) running time. This is done with a auxiliary function pi[1..m] precomputed from P in time O(m).

The key is this auxiliary function pi, which is the prefix function. This function contains knowledge about how the pattern matches shifts against itself. If we know how the pattern matches against itself, we can slide the pattern more characters ahead than just one character as in the naïve algorithm.

Consider this case:

```
P:      ABABAB|AB
T:    ...ABABAB|XZ  ...
```

Suppose that the next character, X, does not match A. The naïve method just moves P one character ahead and tries again. But we can move it farther ahead!

```
P:          |ABAB|ABCB
T:     ...AB|ABAB|X...
```

We can slide the pattern ahead so that the longest PREFIX of P that we have matched, matches the longest SUFFIX of T that we have already matched. Now we can just test the x to see if it matches with the a, and continue on.

Another example:

```
P:      BAAAAAA|A
T:  …  BAAAAAA|XY…
```

The longest prefix of P that matches a suffix of T is nothing, so we can just slide the whole pattern over:

```
P:         BAAAAAAAA
T:      …|XY…
```

How can we precompute the pattern shifts against some text T, when we don't know what T is? We can precompute the pattern shifts, because we know that at some point in P, we have matched T with P, so we really just need to compute prefixes and suffixes within P.

How to compute shifts (later we will store an index to the correct pattern character to compare in pi[]) Given P=ABABACA:

Set shift[1] to 1. If the second character mismatches, it means we should use as the shift the prefix that matches a suffix of the stuff we've already matched. This is only one character, so we should shift the pattern over by one.

Look at first 2 characters in P: AB
Find longest prefix that is a suffix. There isn't one, so set shift[2] to the length of P so far, which is 2. We should shift the pattern by two if the character after B mismatches.

```
Ex:    P=      AB|A
       T=    …AB|C
```

Shift by two:

```
       P=        ABA
       T=     …|C  …
```

Look at first 3 characters in P: ABA
Find longest prefix that is a suffix. It is "A", so set shift[3] to 2. We should shift the pattern by two to line up the first A with the last A if the next character doesn't match.

```
Ex:    P=     ABA|C
       T=…    ABA|B         …
```

Shift by two:

```
       P=          A|BAC
       T=…      ABA|B       …
```

Look at first 4 characters in P: ABAB
Find longest prefix that is a suffix. It is "AB", so set shift[4] to 2. We should shift the pattern by two to line up the first AB with the last AB if the next character doesn't match.

```
Ex:     P=      ABAB|A
        T=…     ABAB|B      …
Shift by two:
        P=        AB|ABA
        T=…     ABAB|B      …
```

Look at first 5 characters in P: ABABA
Find longest prefix that is a suffix. It is "ABA", so set shift[5] to 2. We should shift the pattern by two to line up the first ABA with the last ABA if the next character doesn't match.

```
Ex:     P:       ABABA|B
        T:      …ABABA|A     …
Shift by two:
        P:           ABA|BAB
        T:  …     AB|ABA|A       …
```

Skip to first 7 characters in P: ABABACA
Find longest prefix that is a suffix. It is "A", so set shift[7] to 6. We should shift the pattern by six to line up the first A with the last A if the next character doesn't match.

```
Ex:     P:      ABABACA|D
        T:…     ABABACA|B …
Shift by six:
        P:            A|BABACAD
        T:…     ABABACA|B …
```

Overview of algorithm to construct pi values:

```
pi[1] ← 0
k ← 0
m ← length[P]
For i ← 2 to m
    Find longest prefix $P_i$ that is a suffix within P[1..i]
    set pi[i] ← i - shift for $P_i$          ; Index to last char in pattern
                                             ; that matches text.
                                             ; If zero, nothing left to shift.
```

This algorithm runs in O(m) amortized time.

Once we have the pi values, we can write an algorithm to do the string matching fairly easily:

```
KMP(T,P)
    n ← length[T]
    m ← length[P]
    pi ← Compute-Pi-Values
    q ← 0
    for i ← 1 to n do
        while q>0 and P[q+1]<>T[i] do q ← pi[q]
        if P[q+1]=T[i] then q ← q+1
        if q=m then
            pattern has matched at shift i-m
            q ← pi[q]
```

KMP top level algorithm runs in time O(n), and the call to compute Pi is O(m) so the total runtime is O(m+n).

Other example: P= aaaa
T= aaaaaaaaaaaaa

Shift P along with T to get O(n+m) runtime.

Boyer-Moore Algorithm

Boyer and Moore's algorithms work pretty well for patterns that are long and alphabets that are big. The idea is to compare the pattern from the right to the left, instead of left to the right.

Example:

```
P=   must
T=   if you wish to understand others you must
        |
        compare starting here, work backwards
        since there is no "y" inside P, we can shift it by all 4 characters.
```

```
T=   if you wish to understand others you must
         must
            must
               must
                 must
                     must
                      must
                          must
                            must
```

```
must
   must
```

When the bad character in T is contained within P, we can only shift up to the point where the bad character matches up with the corresponding character in P. This is called the "bad character" heuristic. When we are comparing backwards within P, if we ever run into a mismatch then we just shift the pattern over to compare the leftmost characters in P with the characters we already examined in T.

In this example we only used 18 comparisons! The text is actually 41 characters long, so we can do matching in sublinear time in the best case. Worst case is $O(nm+|\Sigma|)$: requires $O(m+|\Sigma|)$ to compute the last-bad character, and we could run into same worst case as the naïve algorithm (consider P=aaaa, T=aaaaaaaa…). However, on average, Boyer-Moore is often the algorithm of choice.

See implementation in book.