CPSC 490 – Problem Solving in Computer Science

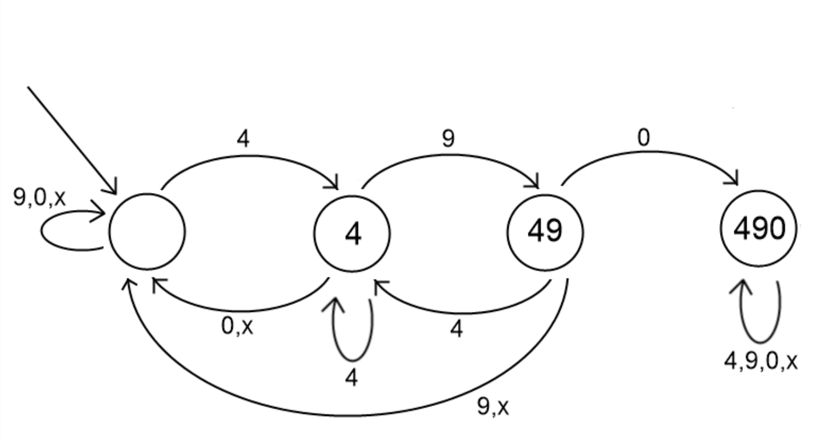
Lecture 9: KMP, Trie

Jason Chiu and Raunak Kumar

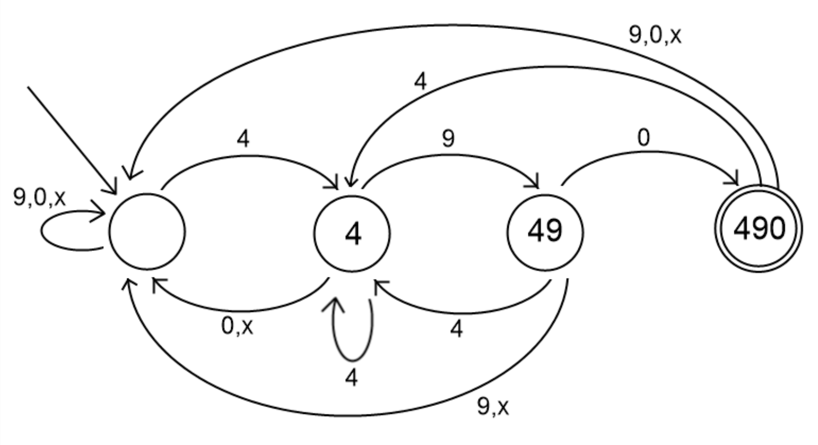
2017/01/25

University of British Columbia

Remember this DFA?



What does this slightly different DFA do?



A new algorithm for string matching

We have discovered a super fast algorithm for string matching!

Suppose we want to find search for all copies of string B in string A

- Somehow construct the DFA for string B quickly
- Run string A through the DFA and output (index - len(B) + 1) for all indices when the DFA is in the "completely matched" state

Can we really do this? What's the problem?

Simplifying the DFA

Problem

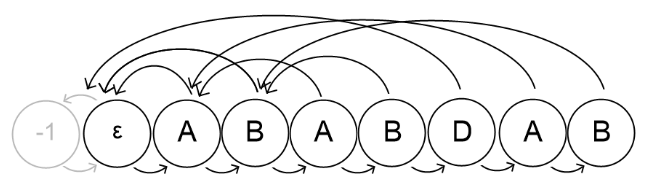
- Size of the DFA proportional to size of alphabet – could be as bad as the size of the search string ($O(m^2)$, bad!)
- The problem is too many transitions

But we can fix this problem!

Simplifying the DFA

Idea: only need two transitions – success and failure!

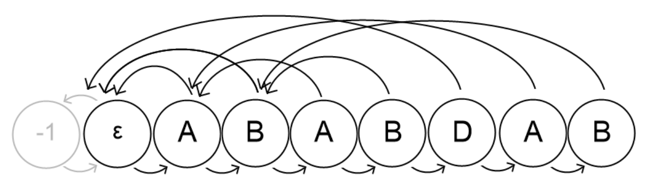
- Success[k] = k+1
- Fail[0] = -1, Fail[k] = next longest prefix that could possibly match, if the current match of length k fails to match next char



Simplifying the DFA

Idea: only need two transitions – success and failure!

- Success[k] = k+1
- Fail[0] = -1, Fail[k] = next longest prefix that could possibly match, if the current match of length k fails to match next char

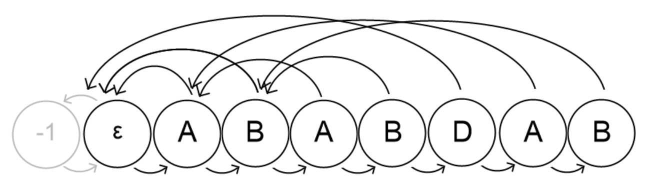


How do we draw the *k*-th Fail arrow?

Simplifying the DFA

Idea: only need two transitions – success and failure!

- Success[k] = k+1
- Fail[0] = -1, Fail[k] = next longest prefix that could possibly match, if the current match of length k fails to match next char

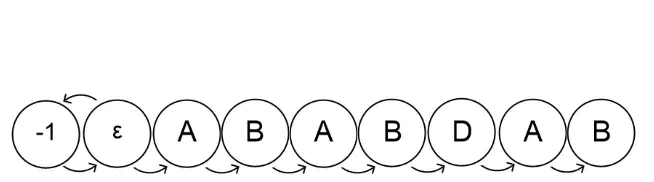


How do we draw the k-th Fail arrow? Follow the $(k-1)$-th Fail arrow <u>at least once</u> and until next char matches or $k = -1$, then move right.

Simplifying the DFA

Idea: only need two transitions – success and failure!

- Success[k] = k+1
- Fail[0] = -1, Fail[k] = next longest prefix that could possibly match, if the current match of length k fails to match next char



How do we draw the k-th Fail arrow? Follow the $(k-1)$-th Fail arrow at least once and until next char matches or $k = -1$, then move right.

Simplifying the DFA

Idea: only need two transitions – success and failure!

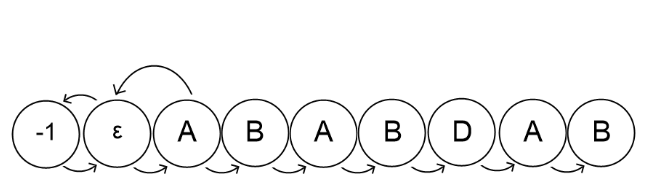
- Success[k] = k+1
- Fail[0] = -1, Fail[k] = next longest prefix that could possibly match, if the current match of length k fails to match next char

How do we draw the k-th Fail arrow? Follow the $(k-1)$-th Fail arrow <u>at least once</u> and until next char matches or $k = -1$, then move right.

Simplifying the DFA

Idea: only need two transitions – success and failure!

- Success[k] = k+1
- Fail[0] = -1, Fail[k] = next longest prefix that could possibly match, if the current match of length k fails to match next char

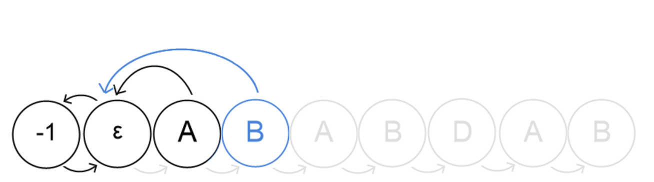


How do we draw the k-th Fail arrow? Follow the $(k-1)$-th Fail arrow <u>at least once</u> and until next char matches or $k = -1$, then move right.

Simplifying the DFA

Idea: only need two transitions – success and failure!

- Success[k] = k+1
- Fail[0] = -1, Fail[k] = next longest prefix that could possibly match, if the current match of length k fails to match next char

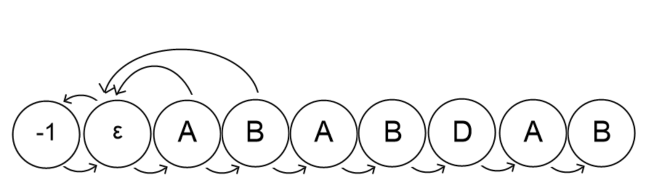


How do we draw the k-th Fail arrow? Follow the $(k-1)$-th Fail arrow <u>at least once</u> and until next char matches or $k = -1$, then move right.

Simplifying the DFA

Idea: only need two transitions – success and failure!

- Success[k] = k+1
- Fail[0] = -1, Fail[k] = next longest prefix that could possibly match, if the current match of length k fails to match next char

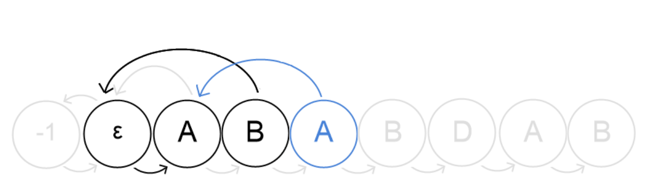


How do we draw the k-th Fail arrow? Follow the $(k-1)$-th Fail arrow <u>at least once</u> and until next char matches or $k = -1$, then move right.

Simplifying the DFA

Idea: only need two transitions – success and failure!

- Success[k] = k+1
- Fail[0] = -1, Fail[k] = next longest prefix that could possibly match, if the current match of length k fails to match next char

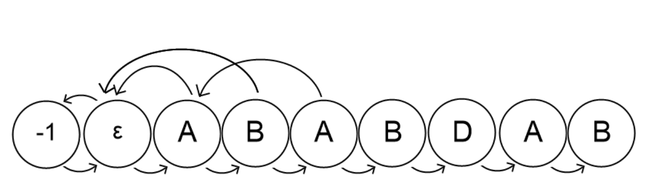


How do we draw the k-th Fail arrow? Follow the $(k-1)$-th Fail arrow at least once and until next char matches or $k = -1$, then move right.

Simplifying the DFA

Idea: only need two transitions – success and failure!

- Success[k] = k+1
- Fail[0] = -1, Fail[k] = next longest prefix that could possibly match, if the current match of length k fails to match next char

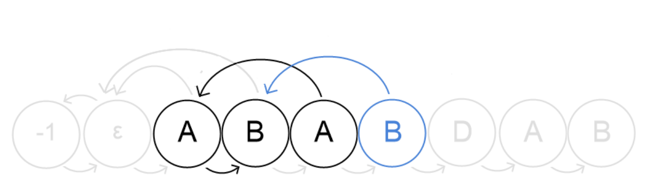


How do we draw the k-th Fail arrow? Follow the $(k-1)$-th Fail arrow <u>at least once</u> and until next char matches or $k = -1$, then move right.

Simplifying the DFA

Idea: only need two transitions – success and failure!

- Success[k] = k+1
- Fail[0] = -1, Fail[k] = next longest prefix that could possibly match, if the current match of length k fails to match next char

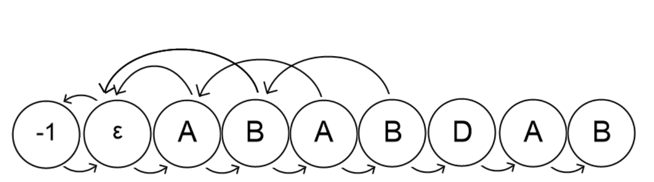


How do we draw the k-th Fail arrow? Follow the $(k-1)$-th Fail arrow <u>at least once</u> and until next char matches or $k = -1$, then move right.

Simplifying the DFA

Idea: only need two transitions – success and failure!

- Success[k] = k+1
- Fail[0] = -1, Fail[k] = next longest prefix that could possibly match, if the current match of length k fails to match next char

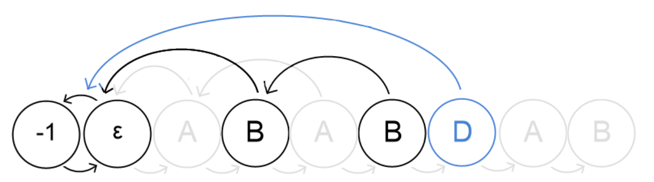


How do we draw the k-th Fail arrow? Follow the $(k-1)$-th Fail arrow at least once and until next char matches or $k = -1$, then move right.

Simplifying the DFA

Idea: only need two transitions – success and failure!

- Success[k] = k+1
- Fail[0] = -1, Fail[k] = next longest prefix that could possibly match, if the current match of length k fails to match next char

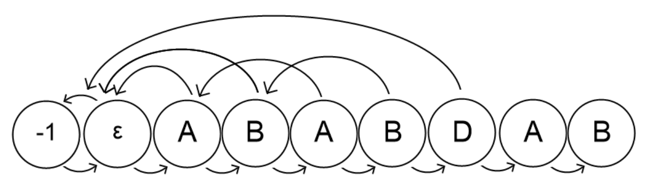


How do we draw the k-th Fail arrow? Follow the $(k-1)$-th Fail arrow <u>at least once</u> and until next char matches or $k = -1$, then move right.

Simplifying the DFA

Idea: only need two transitions – success and failure!

- Success[k] = k+1
- Fail[0] = -1, Fail[k] = next longest prefix that could possibly match, if the current match of length k fails to match next char

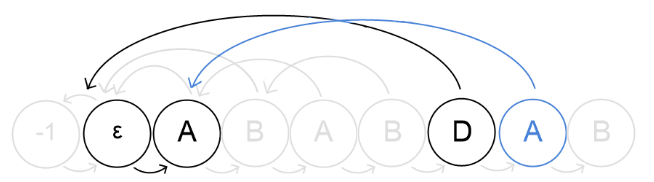


How do we draw the k-th Fail arrow? Follow the $(k-1)$-th Fail arrow at least once and until next char matches or $k = -1$, then move right.

Simplifying the DFA

Idea: only need two transitions – success and failure!

- Success[k] = k+1
- Fail[0] = -1, Fail[k] = next longest prefix that could possibly match, if the current match of length k fails to match next char

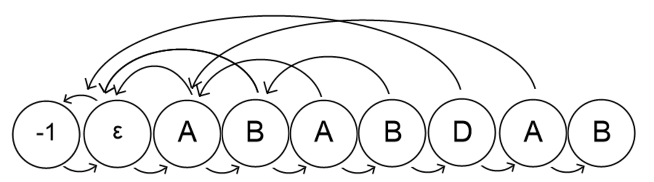


How do we draw the k-th Fail arrow? Follow the $(k-1)$-th Fail arrow at least once and until next char matches or $k = -1$, then move right.

Simplifying the DFA

Idea: only need two transitions – success and failure!

- Success[k] = k+1
- Fail[0] = -1, Fail[k] = next longest prefix that could possibly match, if the current match of length k fails to match next char

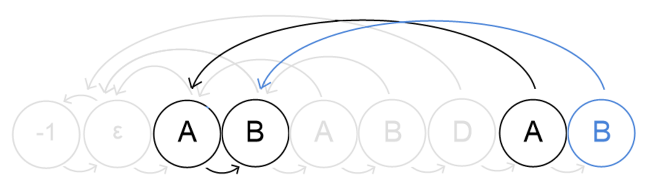


How do we draw the *k*-th Fail arrow? Follow the $(k-1)$-th Fail arrow <u>at least once</u> and until next char matches or $k = -1$, then move right.

Simplifying the DFA

Idea: only need two transitions – success and failure!

- Success[k] = k+1
- Fail[0] = -1, Fail[k] = next longest prefix that could possibly match, if the current match of length k fails to match next char

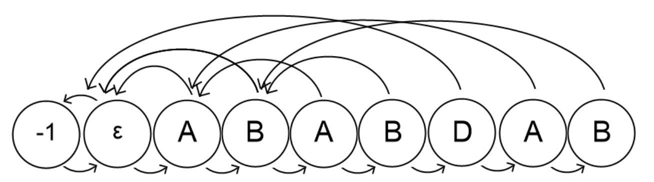


How do we draw the k-th Fail arrow? Follow the $(k-1)$-th Fail arrow <u>at least once</u> and until next char matches or $k = -1$, then move right.

Simplifying the DFA

Idea: only need two transitions – success and failure!

- Success[k] = k+1
- Fail[0] = -1, Fail[k] = next longest prefix that could possibly match, if the current match of length k fails to match next char

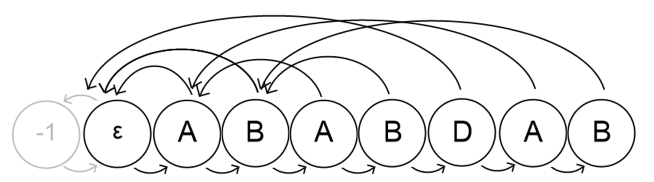


How do we draw the k-th Fail arrow? Follow the $(k-1)$-th Fail arrow <u>at least once</u> and until next char matches or $k = -1$, then move right.

Simplifying the DFA

Idea: only need two transitions – success and failure!

- Success[k] = k+1
- Fail[0] = -1, Fail[k] = next longest prefix that could possibly match, if the current match of length k fails to match next char



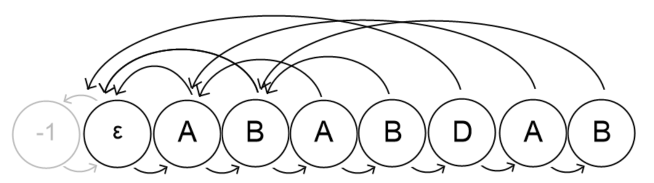
How do we draw the k-th Fail arrow? Follow the $(k-1)$-th Fail arrow <u>at least once</u> and until next char matches or $k = -1$, then move right.

When we see char c, what's next state?

Simplifying the DFA

Idea: only need two transitions – success and failure!

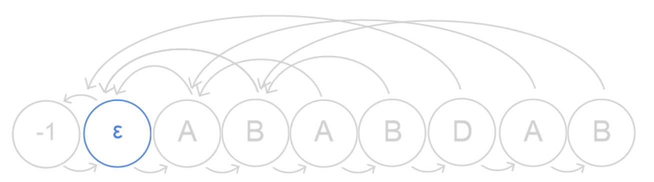
- Success[k] = k+1
- Fail[0] = -1, Fail[k] = next longest prefix that could possibly match, if the current match of length k fails to match next char



How do we draw the k-th Fail arrow? Follow the $(k-1)$-th Fail arrow <u>at least once</u> and until next char matches or $k = -1$, then move right.

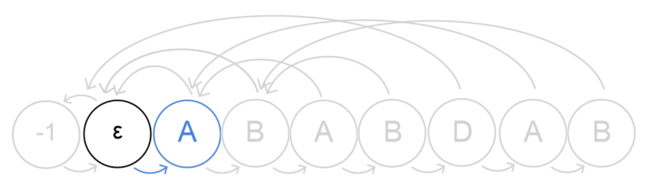
When we see char c, what's next state? Follow Fail arrow until next char matches or $k = -1$, then move right.

String matching with simplified DFA



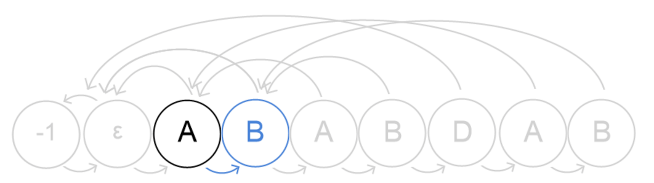
“ABABABDABC”

String matching with simplified DFA



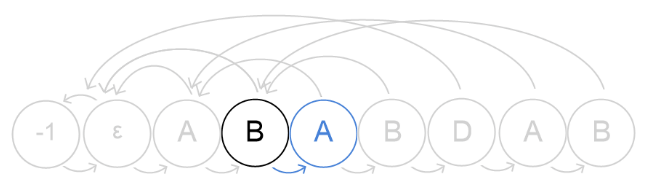
"ABABABDABC"

String matching with simplified DFA



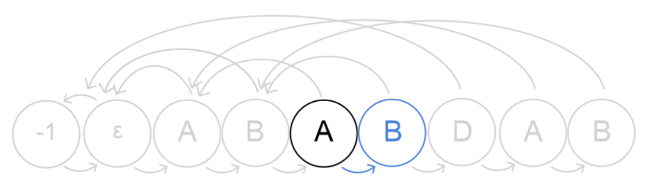
"ABABABDABC"

String matching with simplified DFA



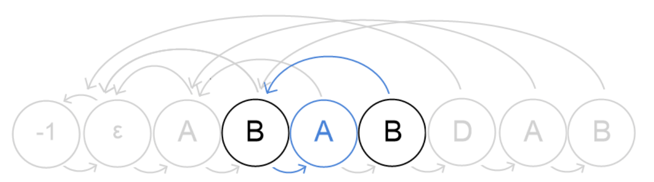
"ABABABDABC"

String matching with simplified DFA



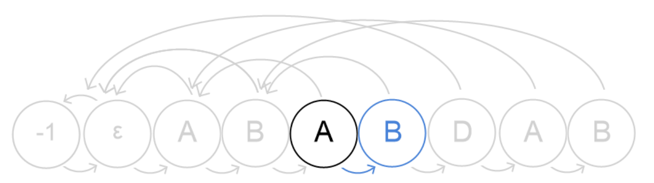
"ABABABDABC"

String matching with simplified DFA



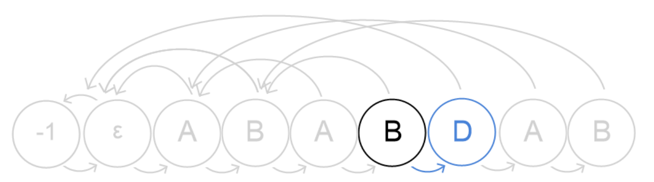
"ABABABDABC"

String matching with simplified DFA



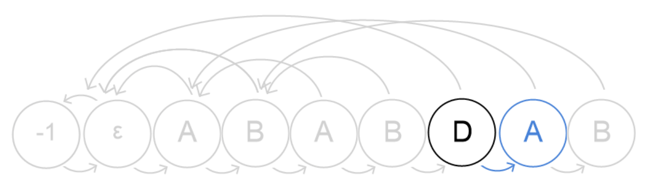
"ABABABDABC"

String matching with simplified DFA



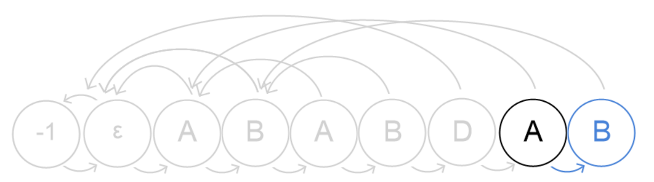
"ABABAB**D**ABC"

String matching with simplified DFA



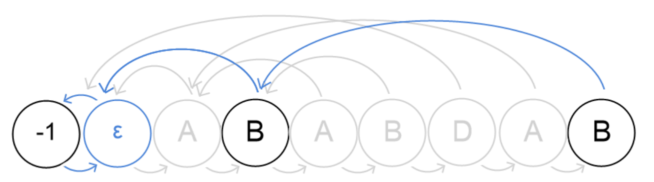
"ABABABDABC"

String matching with simplified DFA



"ABABABDABC"

String matching with simplified DFA



"ABABABDABC"

The KMP Algorithm: Building the DFA

```
KMP_INIT(W):
  initialize array Fail of size |W|+1
  set Fail[0] = -1
  for i in 1 to |W|
    let nxt = Fail[i-1]
    while nxt >= 0 && W[nxt] != W[i-1]:
      nxt = Fail[nxt]
    set Fail[i] = nxt + 1
  return Fail
```

The KMP Algorithm: Matching

```
KMP_MATCH(Fail, W, S):
  initialize cur = 0, matches = empty list
  for i in 0 to |S| - 1
    while cur >= 0 && W[cur] != S[i]:
      cur = Fail[cur]
    cur = cur + 1
    if cur == |W|:
      add i - |W| + 1 to matches
      cur = Fail[cur]
  return matches
```

The KMP Algorithm: Time Complexity Analysis

Building the DFA for search string of size m

- To build next arrow, start at end point of previous arrow, take 0 or more back arrows, and 1 forward arrow.
- Every backward arrow move back at least 1 state
- No arrow moves past -1
- $\Rightarrow$ Backward arrows $\leq$ forward arrows = $O(m)$

The KMP Algorithm: Time Complexity Analysis

Building the DFA for search string of size m

- To build next arrow, start at end point of previous arrow, take 0 or more back arrows, and 1 forward arrow.
- Every backward arrow move back at least 1 state
- No arrow moves past -1
- $\Rightarrow$ Backward arrows $\leq$ forward arrows = $O(m)$

Finding search string in target string of size n

- To get next state, take 0 or more back arrows, 1 forward arrow
- $\Rightarrow$ Similar reasoning gives no more than $O(n)$ arrows

The KMP Algorithm: Time Complexity Analysis

Building the DFA for search string of size m

- To build next arrow, start at end point of previous arrow, take 0 or more back arrows, and 1 forward arrow.
- Every backward arrow move back at least 1 state
- No arrow moves past -1
- $\Rightarrow$ Backward arrows $\leq$ forward arrows = $O(m)$

Finding search string in target string of size n

- To get next state, take 0 or more back arrows, 1 forward arrow
- $\Rightarrow$ Similar reasoning gives no more than $O(n)$ arrows

$\Rightarrow$ Time complexity is $O(m + n)$

Problem 1 – Wildcards

Find at least one occurrence of S_1 in string S_2.

Catch: a $*$ in string A matches any sequence of characters.

S_1	S_2	Match?
aa*b	aab	Yes
	aacdab	Yes
	caaccbd	Yes
	aacdaa	No
	accccb	No

Figure 1: Example of wildcard matching

How many $*$ can you handle?

Problem 1 – Solution

- Cut S_1 by $*$ into pieces $T_1, T_2, \ldots, T_k$
- Find first copy of T_1, then the first copy of T_2 after T_1, and so on
- Time complexity: still $O(m + n)$

Problem 2 – Wildcards Again

What if we only have one wild card, but it can appear anywhere?

S_1	S_2	Match?
computer	a computer	Yes
	cooooomputer	Yes
	compute0r	Yes
	cooompuuuter	No

Figure 2: Example of matching one wildcard anywhere

Problem 2 – Solution

Idea: we need prefix and suffix match information, but suffix = prefix of reverse string!

Problem 2 – Solution

Idea: we need prefix and suffix match information, but suffix = prefix of reverse string!

- Run KMP to find S_1 in S_2, and reverse(S_1) in reverse(S_2)
- Keep track of the KMP state at each character of S_2:
  - $A(k) =$ longest prefix of S_1 that matches suffix of $S_2[0 \ldots k]$
  - $B(k) =$ longest suffix of S_1 that matches prefix of $S_2[k \ldots n-1]$

Problem 2 – Solution

Idea: we need prefix and suffix match information, but suffix = prefix of reverse string!

- Run KMP to find S_1 in S_2, and reverse(S_1) in reverse(S_2)
- Keep track of the KMP state at each character of S_2:
    - $A(k) =$ longest prefix of S_1 that matches suffix of $S_2[0 \dots k]$
    - $B(k) =$ longest suffix of S_1 that matches prefix of $S_2[k \dots n-1]$
- Check if there are indices $i < j$ such that $A(i) + B(j) = m$
    - Can be done in linear time: iterate i from 0 to $n-1$, keep a boolean array of which values $A(i)$ we have seen so far, check if we have seen $m - B(i+1)$

Problem 2 – Solution

Idea: we need prefix and suffix match information, but suffix = prefix of reverse string!

- Run KMP to find S_1 in S_2, and reverse(S_1) in reverse(S_2)
- Keep track of the KMP state at each character of S_2:
  - $A(k) =$ longest prefix of S_1 that matches suffix of $S_2[0 \ldots k]$
  - $B(k) =$ longest suffix of S_1 that matches prefix of $S_2[k \ldots n-1]$
- Check if there are indices $i < j$ such that $A(i) + B(j) = m$
  - Can be done in linear time: iterate i from 0 to $n-1$, keep a boolean array of which values $A(i)$ we have seen so far, check if we have seen $m - B(i+1)$
- Time complexity: $O(m+n)$

Trie

Problem: Implement `Map<String, Value>`

Suppose we store N strings of length $\leq M$ in a balanced BST.

Time complexity: $O(M \cdot \log N)$. Space complexity: $O(MN)$

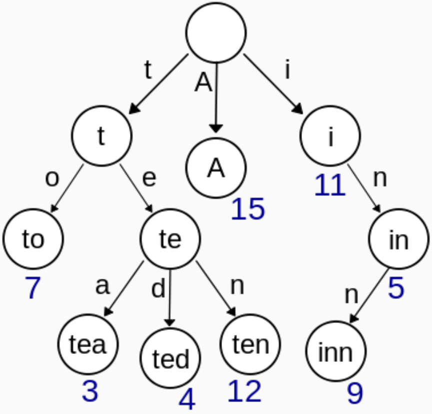
Can't do partial prefix match :(

Can we do better?

A Trie is a Tree!

Observation: there are only 26 letters in the alphabet, so we are storing lots of duplicates! Why not use the alphabet to form a tree?

Keys: to, tea, ted, ten, A, i, in, inn.



Source: *Wikipedia*

Trie Structure and Operations

```cpp
struct TrieNode {
  bool isWord;
  TrieNode *child[26];
  TrieNode() {
    isWord = false;
    memset(child, 0, sizeof child);
  }
};
```

Exercise: code up find(), insert(), delete(), isPrefixMatch()

Time complexity: $O(N)$ per operation where N = max string length
Space complexity: $O(NC)$ for C letters in alphabet

Problem 3 – Word Game

Two player game where you alternate turns adding a letter to a string.

At every turn, the string must be prefix of some word.

The person who adds the last letter of a word loses.

If you go first, can you win? Find the winning strategy!

Problem 3 – Solution

Perform tree dp on the trie of all words

- State: f(node) = can you win if you are here
- f(trie node that is a word) = false
- f(node) = true if f(child) = false for some child
- f(node) = false if f(child) = true for all child

Next Class

Aho Corasick
(i.e. KMP on a Trie)