String matching

CS240 Spring 2020

Rui Fan

Problem and motivation

- Given a text T, find a string S in T.
  - S is often called a “pattern”.
- Ex T=“prefix”, S=“fix” is in T, S’=“six” is not.
- Applications
  - Email, word processor.
  - Search engine.
  - Anti-plagiarism.
  - Sequence alignment.
    - Find a DNA snippet in a genome.

Brute force algorithm

- Compare S against every length |S| substring in T.
  - Green is matched letter, red is mismatch.
  - Shift S in T until all of S matched. If end of T reached, S isn't in T.

```
T   A C A B A B A C A B A B A B C A B
S   A B A B A B C
      A B A B A B C
        A B A B A B C
          A B A B A B C
            A B A B A B C
              A B A B A B C
                A B A B A B C
                  A B A B A B C
                    A B A B A B C
```

Brute force algorithm complexity

- Say T has length n, S has length m $\leq$ n.
  - Comparing S to T at each position takes O(m) time.
  - There are at most n-m shifts.
  - Takes O(m)*(n-m) = O(mn) time total.
- Can we do better? Say O(n) time?
  - O(n) time is optimal, since you have to read every letter in T.

Knuth-Morris-Pratt algorithm

- An O(n) algorithm proposed in 1977.
- Brute force backs up S and T all the way on each mismatch, which makes it slow.
- Sometimes not necessary to back up so far.
- Preprocess S to determine smallest amount to back up on mismatch.
- Supposed we matched j letters of S and T, and we're on the i'th letter of T.
- There's a mismatch between S[j+1] and T[i+1].
  - How far should we back up from the mismatch location?

i

j

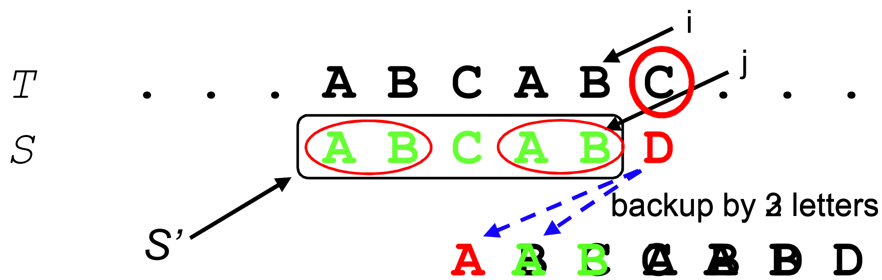
T . . . A B C A B C . . .

S A B C A B D

Backup the start of S from the mismatch location.

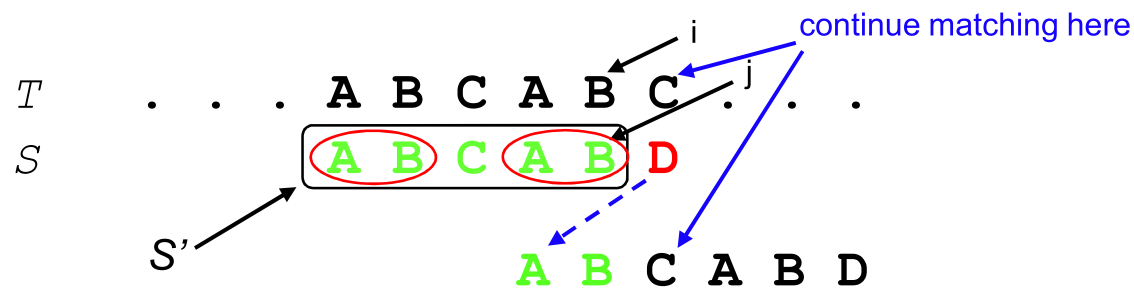
A B C A B D

How much to backtrack



- Goal Continue matching from the mismatch location in T, namely T[i+1].
  - Because we've already processed T up to T[i], so we don't want to process this part again.
- Suppose we back up the head of S by k letters from the mismatch location i+1 in T.
- Observation To achieve the goal, first k letters of S need to match the last k letters of T before T[i+1].
  - Ex If we backup by 2 letters, we can continue matching from the mismatched C in T.
  - Ex If we backup by 3 letters, we get an earlier mismatch, and can't continue matching from the C.

How much to backtrack



- Let S' be the first j letters of S.
  - I.e. it's the part of S we matched to T so far.
  - This is ABCAB is our example.
- Observation If we back up by k letters, then the first k letters of S need to match the last k letters of S'.
  - Last j letters before T[i+1] equal S'.
- Thus, we back up by the largest k, s.t. the first k letters of S match the last k letters of S'.
  - Ex In the example, k=2.
- After backing up, we can continue matching from the i+1'st letter in T and the k+1'st letter in S.

Prefixes and suffixes

`A B C D E F G H I J K`	*string*
`A`	*prefix*
`A B C`	*prefix*
`A B C D`	*prefix*
`            G H I J K`	*suffix*
`                I J K`	*suffix*
`                    K`	*suffix*

- A prefix is a substring including the first letter.
- A suffix is a substring including the last letter.
- A proper prefix or suffix of a string is shorter than the string.
  - Ex ABC is not a proper prefix or suffix of ABC.
- S[1,i] = the length i prefix of S.
  - Ex S = ABCDCAB. S[1,1]=A, S[1,4]=ABCD.

How much to backtrack

- KMP defines a failure function f that specifies how far to back up when there's a mismatch.
  - Suppose we matched first j letters of S to j letters in T, but mismatch on the j+1'st letter.
  - Then we back up f(j) letters from the mismatch in S.
- f(j) equals the length of the longest matching proper prefix and suffix of S[1,j].

```
T     A C A B A B C A B A C
S         A B A B A B C
                 2
f(4)=2        A B A B A B C
```

```
T     A C A B A B A B D A C
S         A B A B A B C
                     4
f(6)=4        A B A B A B C
```

Failure function example

- S=ABABABC.
- f(1)=0: S[1,1]=A. There is no proper prefix or suffix of A.
- f(2)=0: S[1,2]=AB. The only proper prefix and suffix of AB are A, B resp, but these don't match.
- f(3)=1: S[1,3]=ABA. A is a proper prefix and suffix of ABA, and there's no longer matching prefix and suffix.
- f(4)=2: S[1,4]=ABAB. AB is the longest matching proper prefix and suffix of ABAB.
- f(5)=3: S[1,5]=ABABA, longest match is ABA.
- f(6)=4: S[1,6] is ABABAB, longest match is ABAB.
- f(7)=0: S[1,7]=ABABABC, but no proper prefix of S[1,7] matches a suffix.

i	**1**	**2**	**3**	**4**	**5**	**6**	**7**
S	**A**	**B**	**A**	**B**	**A**	**B**	**C**
f	**0**	**0**	**1**	**2**	**3**	**4**	**0**

Implementing KMP

- T has length n, S has length m.
- Assume failure function f has already been computed.

```
i←1,  j←1
while (i ≤ n) and (j ≤ m)
     if T[i] = S[j]
          i ← i+1, j ← j+1
     else if j = 1
          i ← i+1
     else
          j ← f(j-1)+1
if j = m+1
     return i-m
return "S not in T"
```

i is the current character in T, j the current char in S

A match. Move both current chars.

First char in S doesn't match current char in T. So match first char in S against next char in T.

Matched j-1 chars in S and T, but mismatch on j'th char. Backtrack by f(j-1) chars in S. Compare f(j-1)+1'th char in S to T[i].

Found S in T, starting from T[i-m].

Complexity of KMP

- Define the following.
  - A = # times 1st case gets executed.
  - B = # times 2nd case gets executed.
  - C = # times 3rd case gets executed.
- Claim 1 $f(j-1)<j-1$
  - If we match $j-1$ chars and mismatch on j'th, we backup at most $j-2$ chars.
- Claim 2 $A + B < n$
  - Notice i increases when we run A or B.
  - i increases at most $n-1$ times before $i = n$ and we reach the end of T.
- Claim 3 $C \leq A$
  - Only A increases j. C decreases j because $f(j-1)+1<j$.
  - If $j = 1$, then B gets executed.
- Thm 1 KMP has $O(n)$ complexity.
  - The entire while loop runs $< 2n$ times.
    - The number of times it runs is $A+B+C$, and $A+B+C \leq 2A+B < 2n$ by Claims 2,3.

```
while (i ≤ n) and (j ≤ m)
    if T[i] = S[j]
        i ← i+1, j ← j+1
    else if j = 1
        i ← i+1, j ← 1
    else
        j ← f(j-1)+1
```

Implementing the failure function

- S has length m. Compute the values f(1),…,f(m) on S.

```
i←2,  j←1
f(1)←0
while (i ≤ m)
    if S[i] = S[j]
        f(i)← j
        i←i+1,  j←j+1
    else if j > 1
        j ←f(j-1)+1
    else
        f(i) ←0
        i←i+1
```

i is current position in S. j is length of longest matching prefix and suffix of S[1,i-1], plus one.

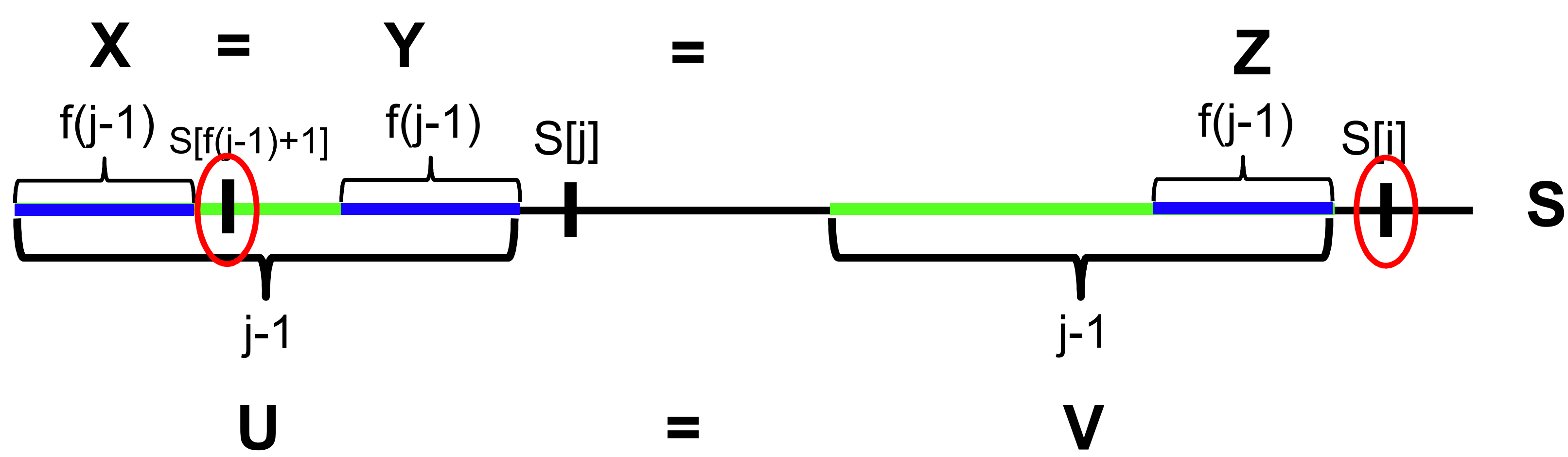
Length j prefix and suffix of S[1,i] match, so set f(i)=j and increment i, j.

Matched j-1 chars, but mismatch on j'th char. What now? See next slide.

There's no matching prefix and suffix of S[1,i]. Set f(i)=0.

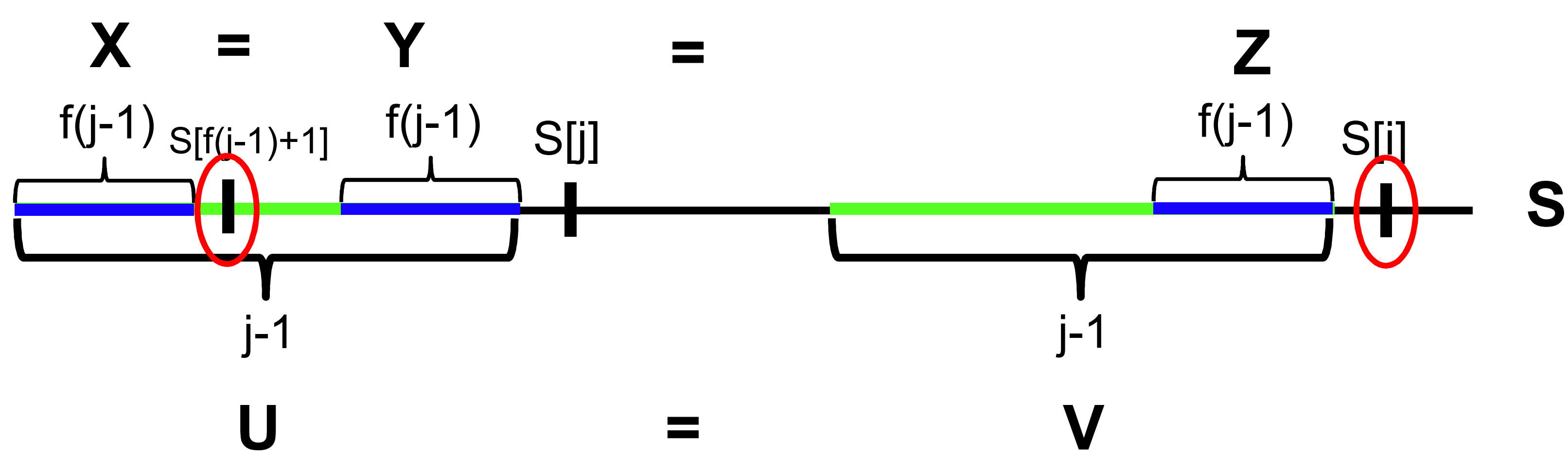
Failure function on mismatch

- Suppose f(i-1)=j-1, i.e. the first j-1 letters of S match last j-1 letters before S[i].
- If S[i]≠S[j], what is f(i)?
- First f(j-1) letters of S match last f(j-1) letters before S[j], by definition of f(j-1).
- So first f(j-1) letters of S match last f(j-1) letters before S[i].
- We try to match the f(j-1)+1'st letter in S to S[i].
  - Set j'=f(j-1)+1.
  - If S[i]=S[j'], then f(i)=j'.
  - If not, then set j'' = f(j'-1)+1, and try to match S[i] and S[j'']. Etc.



Failure function on mismatch

- We want to make sure that when we try to match S[f(j-1)+1] and S[i], we don't miss a longer match.
- Claim The longest matching prefix and suffix of S[1,i] cannot be longer than f(j-1)+1.
- Proof Suppose for contradiction there's a match of length k > f(j-1)+1.
  - Then S[1,k] = S[i-k, i], so S[1, k-1] = S[i-k, i-1].
  - Since U = V, then S[i-k, i-1] = S[j-1-(k-1), j-1] = S[j-k, j-1] = S[1, k-1].
  - So there's a match of length k-1 $\geq$ f(j-1)+1 for a prefix and suffix of S[1, j-1], which is a contradiction to the definition of f(j-1).



Complexity of failure function

- Define the following.
  - A = # times 1st case gets executed
  - B = # times 2nd case gets executed
  - C = # times 3rd case gets executed.
- Claim 1 $A + C < m$.
  - i increases when we run A or C, and i's at most m.
- Claim 2 $B \leq A$.
  - Only A increases j by 1.
  - $f(j-1) < j-1$, because we consider proper prefixes and suffixes
  - So B decreases j by at least 1.
  - If j=0, we run C.
- Thm 2 The FF alg has O(m) complexity.
  - The complexity of FF is O(A+B+C).
  - $A + B + C < 2A + C < 2m$ by the claims.

```
while (i ≤ m)
    if S[i] = S[j]
        f(i)← j+1
        i←i+1,  j←j+1
    else if j > 0
        j ←f(j-1)+1
    else
        f(i) ←0
        i←i+1
```

Example

i	1	2	3	4	5	6
S	A	B	A	B	B	A
f	0	0	1	2	0	1

```
T   A A B A B A A B A B A B B A A B
S   A B A B B A                       f(1) = 0
      A B A B B A                     f(4) = 2
          A B A B B A                 f(3) = 1
              A B A B B A             f(1) = 0
                A B A B B A           f(4) = 2
                    A B A B B A       Match!
```

Complexity of string matching

- Thm Finding a string of length m in a text of length m takes time O(m + n) = O(n) using KMP and FF.
- Boyer-Moore is another O(n) time string matching algorithm.
  - In fact, it works in O(n/m) on average.
  - It's the most efficient algorithm in practice.
- KMP and BM preprocess the string in O(m) time, and search a text in O(n) time.
  - These algorithms are used when the string and text are given online.
- When the text is fixed, but strings are given online, we can preprocess the text for faster searches.
  - Ex Search for words (strings) in Shakespeare's plays (text).
  - Ex Search for snippets of DNA in a reference genome.
- Suffix trees preprocess the text in O(n) time.
  - After this, any string can be searched in O(m) time, instead of O(n) time.
- Suffix arrays, BWT and FM index, many more.
  - See http://www-igm.univ-mlv.fr/~lecroq/string/index.html