

Data Structures and Algorithms II

Lecture 3:

String Search Algorithms

Two ways to speed things up:

- Knuth-Morris-Pratt (KMP) Algorithm - Using knowledge of how string repeats itself.
- Boyer-Moore Algorithm - Using knowledge of where characters occur in search string.

1

```
s1= "aaaaba "  
    "aaaba "  
    * * *
```

Or consider the example:

```
s1= "aabaaaba "  
s2= "aaaba "  
    * * *
```

- The first three characters in s2 are the same.
- s2[2] *failed* to match s1[2].
- Therefore s2[1] and s2[0] can't match s1[2].
- So it's not even worth trying i=1,j=0 or i=2,j=0.
- So we should next try i=3, j=0.

```
s1= "aabaaaba "  
    "aaaba "  
    *
```

NOTE: Simplified technique below only takes into account MATCHES, not mismatches, so doesn't handle above example.

3

Knuth-Morris-Pratt (KMP) Algorithm

Brute-Force algorithm can be inefficient for strings that repeat themselves.

```
s1= "aaaaba "  
s2= "aaaba "  
    * * *
```

Brute force algorithm then matches s2[1] and s1[2].

```
s1= "aaaaba "  
    "aaaba "  
    * * * * *
```

Can we somehow use the fact that the first three characters are the same to skip some of the search?

If we can pre-analyse s2, then we can say:

- The first three characters in s2 are the same.
- s2[0] matched s1[0], s2[1] matched s1[1], s2[2] matched s1[2].
- So s2[0], s2[1] MUST match s1[1], s1[2].
- .. so can start checking from s2[2], saving two comparisons (ie, i=3,j=2)

2

KMP Algorithm

KMP Algorithm is based on constructing a table which answers :

"If string and doc match until char j in string, but then mismatch, how far can we move the string up?"

This table can be constructed by comparing the search string *against itself*. This is valid as in the above we assume that we have a *matching* part of a string.

What we record is *the position in s2 that we can back up to, keeping the position in s1 constant, when a mismatch occurs*. Stored in array next[i].

4

Example: Constructing the table

Where should we back up to in string below if mismatch at $i=3$?

```
x      x = assume mismatch occurred here
aaa??
aaaba
```

Can continue search at..

```
aaa??
aaaba
*
```

Now matching 3rd char in s_2 , so:

```
next[3]=2 (3rd char at posn 2)
```

5

KMP Table Example

Consider $s_2="ababbaa"$.

next[1]?

```
a????? mismatch at char 1
ababbaa
```

```
a?????? Now ``move up`` so
ababbaa comparing 0th character in s2.
next[1]=0
```

next[2]?

```
ab????? mismatch at char2
ababbaa
```

```
ab????? again, can move up so comparing
ab???? 0th char: next[2]=0
```

```
aba???? mismatch at char 3
ababbaa
```

```
aba???? move up so looking at 1st char
ababbaa next[3]=1
```

7

KMP Constructing the Next table

More formally:

We find $next[j]$ by sliding forward the pattern along itself, until we find a match of the first k characters with the k characters before (and not including) position j .

Or..looking for $next[j]$ we find the first index k such that $s_2[0..k-1] = s_2[j-k..j-1]$, e.g:

```
'ababbaa'    s2[0..1] = s2[2..3]
'aba....'    so next[4] = 2.
  ^
```

6

So..

j	$s_2[j]$	$next[j]$
0	a	-1
1	b	0
2	a	0
3	b	1
4	b	2
5	a	0
6	a	1

8

KMP Algorithm

Once we have the "next" table the KMP algorithm is fairly trivial. Set next[0]=-1.

```
i=0; j=0;
while((i<strlen(s1)) && (j<strlen(s2)))
  if(s2[j]==s1[i]) {i++; j++;}
  else j=next[j];
if(j==strlen(s2)) cout << "found at " << i-j;
```

e.g.,

```
abababbaa
ababbaa
*****      next[4]=2
```

```
abababbaa
  ababbaa
    *****
```

9

Boyer-Moore Algorithm

Boyer-Moore is an algorithm which makes ordinary text search much more efficient by:

- Analysing string before search (as KMP).
- Storing, for all possible characters, where they first (right to left) appear in string.
- Using this table to quickly find potential matches.

```
s2='date'
```

```
index[d]=0
index[a]=1
index[t]=2
index[e]=3
```

```
index[anything else] = -1
```

11

KMP Efficiency

KMP can give good efficiency gains for strings with much repetition (e.g., binary files; graphics formats). Less useful for text strings.

Worst case: $N+M$.

Average case: N

cf brute force: worst case $N*M$.

10

Boyer-Moore Algorithm

String s_2 matched from *right to left* (but moved from left to right up document).

Whenever mismatch occurs, look up index value of character c in document causing problem, so find out where/if it occurs in string.

Then move up string so 'index[c]th' character of string is below c , and start searching again from right.

```
s1="some date"
s2="date"
**      m<>t.. index[m]=-1 so
          move so -1th posn of s2 below m
```

```
s1="some date"
  "date"  a<>e.. index[a]=1 so
          *   move so char 1 of s2 below a.
```

```
s1="some date"
  date
  ****
```

12

Boyer-Moore Example

Very often can search very quickly using this method:

```
data structures and algorithms
algorithms
  *
    algorithms
      *
        algorithms
          *****
```

Apart from final comparison at end, only N/M comparisons needed.

(NOTE: Occasionally the above - slightly simplified - technique will result in search string moving to right, not left! So need to add a check for this in implementation).

13

Summary

- String search algorithms made much more efficient by analysing search string before hand.
- KMP analyses how string repeats itself. Boyer-Moore analyses where characters occur in search string.
- KMP good for binary/graphics files, Boyer-Moore good for text files.

15

Efficiency

- If a given letter *usually* doesn't occur in a string, only need approx N/M character comparisons (N =length doc, M =length string).
 - This criteria is met if:
 - length of string less than size of *alphabet* (and fairly even distribution of letters).
- Normal searches of English text meet this criteria (e.g., 10 chars in string, 26+ in alphabet).
- Worst case $N+M$ comparisons.

14