

COMP261 Lecture 18

Marcus Frean

String Searching 2 of 2



Victoria

UNIVERSITY OF WELLINGTON

*Te Whare Wānanga
o te Ūpoko o te Ika a Māui*



CAPITAL CITY UNIVERSITY

String search - recap

- Simple search
 - Slide the window by 1
 - $k = k + 1;$
- Knuth-Morris-Pratt (KMP)
 - Slide the window faster
 - $k = k + i - M[i]$
 - is there a “suffix == prefix”?
 - If No, skip these characters altogether (big jump ahead for S)
 - » $M[i] = 0$
 - If Yes, reuse: no need to recheck those characters!

(smaller jump for S, but start further along it)

 - » $M[i]$ is the length of the “reusable” suffix

$k = 0$
 ↓
 abbabbtabbarsaa;ldifewskf
 abbabb**c**zz
abbabbczz
abbabbczz
 abb**a**bbczz
 slow...

abbabbtabbarsaa;ldifewskf
 abbabb**c**zz
 abb**a**bbczz
 faster...
 abba**b**bczz
 abbab**a**bczz
 ↗ i


Knuth-Morris-Pratt (KMP) algorithm

After a mismatch, advance to the earliest place where search string could possibly match.

- avoids the re-checking of characters that brute-force does

How far can we advance safely? (as much as possible, no further)

- Use a table based on the search string.
- Let $M[0..m-1]$ be a table showing how far to back up the search if a prefix of S has been matched.

 m is the length of S

KMP - how far to move along? (in general)

- long text: . . . ananx??? . . .
- string: anan**c**ba
- If mismatch at string position s (and text position $k+i$)
 - find longest **suffix of text** (up to just before the fail point) that matches a **prefix of string**
 - move k forward by $(i - \text{length of substring})$
 - keep matching from $i \leftarrow \text{length of substring}$
- special case:
 - if $i = 0$, then move k to $k + 1$ and match from $i \leftarrow 0$

KMP

fail: not 'g'

- **an**zn#????????????????????
anzn**g**fg

- having got to a fail point, where should we check next?
- jump ahead, and re-start at... where?

the fail point

fail: not 'g'

but it could be 'a'!

- **an**an#????????????????????
anan**g**fg

- what about this one?
- unsafe to jump straight to the fail point!

move S by 2, but restart
from the fail point (#)

- **an**an#????????????????????
anan**a**fg

- what about this one?
- (nb: in theory, could jump further in such cases, for a small extra saving)

simplest: treat
same as above

KMP

MOVING FROM THE LEFT of the search string S, on mismatch with T we check for a suffix == prefix, skip ahead that many, and continue checking matches from the fail point.

T: abbabb**t**abbabbczzrsaldifewsk
S: abbabb**c**zz

suffix of 3 **in the matched part**:
skip ahead 3, and restart from “**t**”

T: abbabb**t**abbabbczzrsaldifewsk
S: **abb**abbczz

no suffix: move to “**t**”, and restart

T: abbabb**t**abbabbczzrsaldifewsk
S: **a**bbabbczz

no suffix: move to “**t**”, and restart

T: abbabb**t**abbabb**c**zzrsaldifewsk
S: abbabb**c**zz

and we could precompute
all these jumps, just from S

Knuth Morris Pratt, the algorithm

input: string $S[0 \dots m-1]$, text $T[0 \dots n-1]$, jump table $M[0 \dots m-1]$

output: the position in T at which S is found, or -1 if not present

variables: $k \leftarrow 0$ *start of current match in T*
 $i \leftarrow 0$ *position of current character in S*

```

while  $k + i < n$ 
    if  $S[i] = T[k + i]$  then      // match at  $i$ 
         $i \leftarrow i + 1$ 
        if  $i = m$  then return  $k$       // found  $S$ 
    else if  $M[i] = -1$  then      // mismatch, no self overlap
         $k \leftarrow k + i + 1, i \leftarrow 0$ 
    else      // mismatch, with self overlap
         $k \leftarrow k + i - M[i]$       // match position jumps forward
         $i \leftarrow M[i]$ 

return -1      // failed to find  $S$ 
  
```

detour to slide 10 here,
then return here.

How do we build the “jump” table? Example.

- Consider the search string `abcdabd`.
- Look for a proper suffix of failed match, which is a prefix of `S`, starting at each position in `S`
 - so suffix ends at previous position.

- 0 : `abcdabd`

We can't have a failed match at position 0.

Special case, set `M[0]` to -1.

- 1 : `abcdabd`

a not a proper suffix.

Special case, set `M[1]` to 0.

- 2 : `abcdabd`

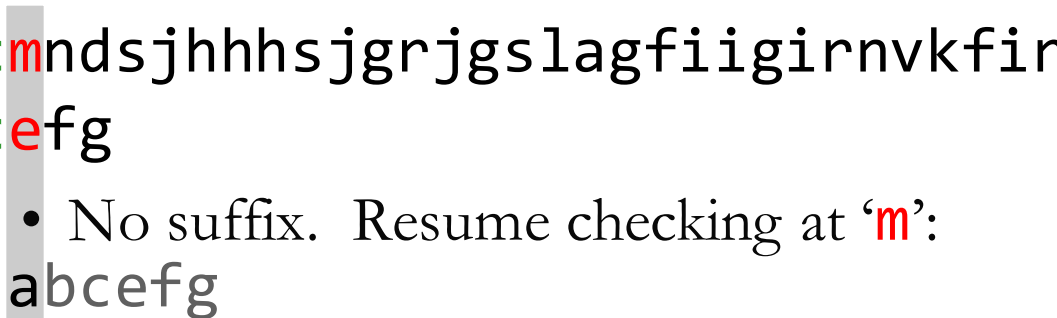
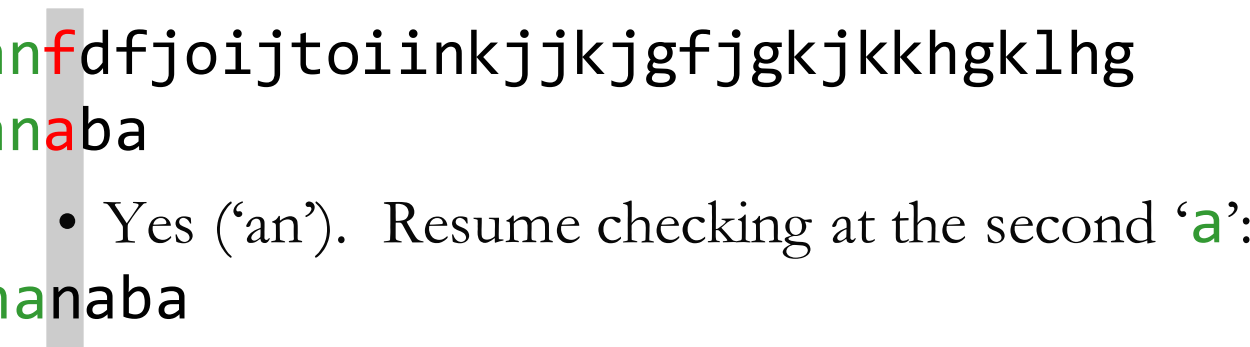
b not a prefix, set `M[2]` to 0.

How do we build the “jump” table? Example.

- 3 : abc**d**abd
abc has no suffix which is a prefix, set $M[3]$ to 0.
- 4 : abcdabd
abcd has no suffix which is a prefix, set $M[4]$ to 0.
- 5 : abcdabbd
a is longest suffix which is a prefix, set $M[5]$ to 1.
- 6 : abcdabd
ab is longest suffix which is a prefix, set $M[6]$ to 2.
- Knowing what we matched before allows us to determine length of next match.

How do we precompute the “jump” table, M?

Look for *suffix of a failed match* which is *prefix of the search string*. eg:

- 
 - No suffix. Resume checking at ‘m’:
abcefg
- 
 - Yes (‘an’). Resume checking at the second ‘a’:
ananaba
- NB: suffix of a partial match is also part of the search string...
We can find partial matches just by analysing the search string!

KMP – Partial Match Table

Index	0	1	2	3	4	5	6
S	a	b	c	d	a	b	d
M	-1						

KMP – Partial Match Table

Index	0	1	2	3	4	5	6
S	a	n	a	n	a	b	a
M	-1						

Building the table.

input: $S[0 \dots m-1]$ // the string
output: $M[0 \dots m-1]$ // match table

M:	0	1	2	3	4	5	6	7
								.

initialise: $M[0] \leftarrow -1$
 $M[1] \leftarrow 0$
 $j \leftarrow 0$ // position in prefix
 $pos \leftarrow 2$ // position in table

→
 andandba
 andandba
 →

while $pos < m$
 if $S[pos - 1] = S[j]$ // substrings $\dots pos-1$ and $0..j$ match
 $M[pos] \leftarrow j+1,$
 $pos++, j++$
 else if $j > 0$ // mismatch, restart the prefix
 $j \leftarrow M[j]$
 else // $j = 0$ // we have run out of candidate prefixes
 $M[pos] \leftarrow 0,$
 $pos++$

String search: Knuth Morris Pratt

- Cost?
- What happens for the worst case for brute force search?

S = aaaaab

T = aaaaaaaaaaaaaaaaaaaaaa

String search: can we do even better?!

- The previous lecture said: “ideally, we’d have an algorithm that never needs to re-trace its steps in the long string. Can we check each letter just once?” (Answer: yes, it’s KMP).

fail

- aaba**h**????????????????????????????????

aaba**a**cb


- but notice **h** is *nowhere* in the key string, so we can jump past...
- Boyer-Moore exploits this notion to the absolute max, so much so that it does *better* than our “aim” of only checking everything once!

String search: Boyer-Moore (details not examinable)

- KMP searches forwards, and gets worse as the search sequence gets longer.
 - It seems implausible that one could do better than looking at each T element only once, and yet...
 - Boyer-Moore algorithm searches backward, gets *better* as search sequence gets longer!
1. Bad character rule – tries to turn mis-match into match
 2. Good suffix rule – tries to keep existing matches okay

Boyer Moore's “Bad Character rule”

(details not examinable)

Go *FROM THE RIGHT* within the search string S, CCTTTT**TGC**


upon a mis-match, we skip until either

- mismatch becomes a match, or
- S moves past the mis-match character

T: GCTT**C**TGCTACCTTTTGC GCGCGCGCGGAA

S: CCTTT**T**GC


T: GCTTCTGCT**A**CTTTTGC GCGCGCGCGGAA

S: CCTTTT**G**C


T: GCTTCTGCTAC**CTTTTGC** GCGCGCGCGGAA

S: CCTTTTGC

Boyer Moore's “Good Suffix rule”

(details not examinable)

Let t be the substring matched by the inner loop.

On mismatch we skip until either

- no mismatch between S and t , or
- S moves past t

T: CGTGCCTACTTACTTACTTACTTACTTACGCGAA
 CGTACCTAC

T: CGTGCCTACTTACTTACTTACTTACTTACTTACGCGAA
 CTACTTAC

T: CGTGCCCTACTTACTTACTTACTTACTTACTTACTTACGCGAA
 CTTACTTAC

Boyer-Moore algorithm

(details not examinable)

This is the go-to algorithm for fast string search in most practical cases.

- At each step, look up *both* jumps, and take max!

T: C T T A T A G C **T** G A T C G C G G C G T A G C G G C G A A
S: G T A G C G G C **G**

bad character: 6

T: C T T A T A G C T G A T **C** **G** **C** **G** G C G T A G C G G C G A A
S: G T A G C **G** **G** **C** **G**

good suffix: 2

T: C T T A T A G C T G A T **C** **G** **C** **G** **G** **C** **G** T A G C G G C G A A
S: G T **A** **G** **C** **G** **G** **C** **G**

good suffix: 7

T: C T T A T A G C T G A T C G C G G C **G T A G C G G C G A A**
S: C T T A T A G C T G A T C G C G G C **G T A G C G G C G**

completely ignored!!