Data Compression 2 Extension:
String Search

Fang-Lue Zhang

String search

"Given a string S and a text T,
look for an occurrence of S as a substring of T"

- Which one? (first, all…)
- What do I do when I find it?

- If found, return index of first character of S in T;
  otherwise return -1 (or some other index outside of T).

- What would you expect the cost to be?

String search - some variations

- Just check whether it's there, returning Boolean.
- Find first/last/any occurrence of S in T.
- Find all occurrences of S in T.
  - What if occurrences overlap?
- Find occurrence(s) as a whole word/anywhere?
- Find occurrences within lines/allow occurrences to extend across line breaks?
- Assume random data? English text? Other data?

```
qwerxcvvtewfzxcfasfed
rsadfsdacfasdrtvtewqw
ertcsvte
wfvtxqwfczsrdcvvtzfec
eeaeszxccvvtvtsafsers
dxzcvtedfaevsadcv
vtvtewfvtxqwfczsvzxgv
tasfvtcasrfvtewqtrwtr
avtecvvtwfxtrac
```

String search

- In Java, we can do this by using:
  - T.indexOf(S);
  - T.lastIndexOf(S);
  - T.contains(S);

- But we'd like to know if these are good choices
  - or if we can do better.

- Let's start with a simple algorithm, and see how we can improve upon it.

Brute force approach

- string: S = ananaba
- text: T = bannabanabananaban
- Look for S, starting at T[0]:

```
ananaba
bannabanabananaban
```

- Look for S, starting at T[1]:

```
 ananaba
bannabanabananaban
```

- Look for S, starting at T[2]:

```
  ananaba
bannabanabananaban
```

- Etc. till found, or none left.

Brute force algorithm

- Basic idea:
  Look for S in T,
  starting at positions T[0], T[1], ….

- What is last position in T we need to consider?

```
for  k ← 0  to  T.length() - S.length()
     if  T.substring(k, S.length()).equals(S) then return k
return   -1
```

Brute force algorithm

Can we improve?

```
for  k ← 0  to  T.length() - S.length()
   if  T.substring(k, S.length()).equals(S) then return k
return   -1
```

- First, some very simple "improvements":
  - **Don't call `length` methods in the loop**.
    Avoid cost of method call (compiler may inline it).
  - **Don't call `substring` method in the loop.**
    Don't need to copy the substring to a new string to compare with S.

Brute force algorithm

Assigning `m ← S.length()` and `n ← T.length()` first:

```
• for  k ← 0  to  n-m
      found ← true
      for  i ← 0  to  m-1
         if   S[i] != T[k+i]  then  found ← false,  break
      if  found  then return k
   return   -1
```

- Okay what is the cost?

Brute force algorithm: cost

- `S = s0   s1   s2   s3   …` (m of these)
  `T = t0   t1   t2   t3   t4   t5   t6   t7  …`(n of these)

- What is best/worst/expected cost?

- What if text is random? English?

- What <u>case</u> gives best/worst cost (for any m and n)?
  - How many positions in T need to be considered?
  - How many characters need to be considered at each position?

Brute force algorithm: best cost

- `S = s0  s1  s2  s3` … (m of these)
  `T = t0  t1  t2  t3  t4  t5  t6  t7` …(n of these)

- Suppose `s0` doesn't occur in T.
  - `s0` will be compared to `t0, t1, …`
  - So cost will be?

- Suppose S is a prefix of T.
  - Will compare `s0` with `t0`, `s1` with `t1`, …
  - So cost is?

Brute force algorithm: worst cost

- `S = s0  s1  s2  s3` … (m of these)
  `T = t0  t1  t2  t3  t4  t5  t6  t7` …(n of these)

What case will force the algorithm to do the most comparisons?

- *Hint 1*: Want S not in T, so it tries the maximum number of positions.
- *Hint 2*: At each position, want algorithm to do the **most possible comparisons before failing**.
  →→ Fail on the last character in S!

What inputs would do this?

- What about

```
S = aaaaab
T = aaaaaaaaaaaaaaaaaaaaaa
```

What is the cost?

Would this ever happen with English text?

What sort of data then?

String search: can we do better?

- 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?

fail

- abcdh???????????????????????????
  abcdefg

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

  – this could speed up search a lot!
  – is it "safe"?

String search: can we do better?

fail

- anzn#???????????????????????????
  anzngfg
  - having got to a fail point, where should we check next?
  - jump ahead, and re-start at… where?
- anan#???????????????????????????
  ananafg
  - what about now?
  - It is unsafe to jump to the fail point
- Key idea of KMP algorithm: Use characters in partial match to determine where to start next match attempt.

String search: Example

- T = abc_abcdab_abcdabcdabde
  S = abcdabd

- T = abc_abcdab_abcdabcdabde
  S =   abcdabd

- T = abc_abcdab_abcdabcdabde
  S =   abcdabd

- T = abc_abcdab_abcdabcdabde
  S =   abcdabd

continued next slide

String search: Example

- ```
  T = abc_abcdab_abcdabcdabde
  S =        abcdabd
  ```
- ```
  T = abc_abcdab_abcdabcdabde
  S =          abcdabd
  ```
- ```
  T = abc_abcdab_abcdabcdabde
  S =           abcdabd
  ```
- ```
  T = abc_abcdab_abcdabcdabde
  S =               abcdabd
  ```

Knuth-Morris-Pratt (KMP) algorithm

- The “Knuth” here is Donald Knuth – https://en.wikipedia.org/wiki/Donald_Knuth

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

- never has to re-check a character

How far can we advance safely?

- 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.

String search

- Simple search
  - Slide the window by 1
    - `t = t+1;`
- KMP
  - Slide the window faster
    - `t = t + s - M[s]`
  - Never re-check the matched characters
    - If there is a “suffix ==prefix”?
      - No, skip these characters
        - » `M[s] = 0`
      - Yes, reuse, no need to recheck these characters
        - » `M[s]` is <u>the length of the “reusable” suffix</u>

```
abcdmndsjhhhsjgrjgslagf
abcdefg
```

```
ananfdfjoijtoiinkjjkjgghfj
anangbgba
```

Knuth Morris Pratt

input: string S[0 .. m-1] , text T[0 .. n-1], partial match table M[0 .. m-1]

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

variables: k ← 0 *start of current match in T*

i ← 0 *position of current character in S*

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

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

String search - recap

- Simple search
  - Slide the window by 1
    - `t = t+1;`
- Knuth-Morris-Pratt (KMP)
  - Slide the window faster
    - `t = t + s – M[s]`

```
abbabbtabbarsaa;ldifewskf
abbabbczz
 abbabbczz
  abbabbczz
   abbabbczz
```

slow….

```
abbabbtabbarsaa;ldifewskf
abbabbczz
   abbabbczz
       abbabbczz
           abbabbczz
```

faster….

- is there a "suffix ==prefix"?
  - If No, skip these characters altogether (big jump ahead for S)
    - » `M[s] = 0`
  - If Yes, reuse: <u>no need to recheck those characters</u>!
    (smaller jump for S, but start further along it)
    - » `M[s]` is the length of the "reusable" suffix

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

- long text: `...ananx???.....`
- string: `anancba`

- If mismatch at string position s (and text position t+s)
  - find longest suffix of text (up to just before the fail point) that matches a prefix of string
  - move k forward by (i – length of substring)
  - keep matching from i ← length of substring
- special case:
  - if i = 0, then move k to k + 1 and match from i ← 0

KMP

fail: not 'g'

- anzn#?????????????????????????
  anzngfg
  - 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'!

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

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

- anan#?????????????????????????
  ananafg
  - what about this one?
  - (nb: in theory, could jump further in such cases a 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:    abbabbtabbabbczzrsaldifewsk
S:    abbabbczz
```

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

```
T:    abbabbtabbabbczzrsaldifewsk
S:       abbabbczz
```

no suffix: move to "**t**", and restart

```
T:    abbabbtabbabbczzrsaldifewsk
S:          abbabbczz
```

no suffix: move to "**t**", and restart

```
T:    abbabbtabbabbczzrsaldifewsk
S:           abbabbczz
```

and we could <u>precompute</u>
all these jumps, just from S

KMP, the algorithm

input: string S[0 .. m-1] , text T[0 .. n-1], jump table M[0 .. m-1]
output: the position in T at which S is found, or -1 if not present
variables: k ← 0 *start of current match in T*
i ← 0 *position of current character in S*

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

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

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:` a**<u>b</u>**cdabd
  a not a proper suffix.
  Special case, set M[1] to 0.
- `2:` ab**<u>c</u>**dabd
  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: abcd`**a**`bd`
  abcd has no suffix which is a prefix, set M[4] to 0.
- `5: abcda`**b**`d`
  a is longest suffix which is a prefix, set M[5] to 1.
- `6: abcdab`**d**
  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:

- `abcmndsjhhhsjgrjgslagfiigirnvkfir`
  `abcefg`
  - No suffix. Resume checking at 'm':
    `abcefg`
- `ananfdfjoijtoiinkjjkjgfjgkjkkhgklhg`
  `ananaba`
  - 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	b	c	d	a	b	d
M	-1	0	0	0	0	1	2

KMP – Partial Match Table

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

KMP – Partial Match Table

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

Building the table

M: 0 1 2 3 4 5 6 7

input: S[0 .. m-1] *// the string*

output: M[0 .. m-1] *// match table*

andandba

andandba

```
initialise: M[0] ← -1,  M[1] ← 0
            j ← 0              // position in prefix
            pos ← 2            // position in table

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

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

- `aabah???????????????????????????`
  `aabaacb`

  – 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

- 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, upon a mis-match, we skip until either:

CCTTTTGC

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

T: GCTTCTGCTACCTTTTGCGCGCGCGGAA

S: CCTTTTGC

T: GCTTCTGCTACCTTTTGCGCGCGCGGAA

S: CCTTTTGC

T: GCTTCTGCTACCTTTTGCGCGCGCGGAA

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
T:  CGTGCCTACTTACTTACTTACTTACGCGAA
S:  CTTACTTAC

             t
T:  CGTGCCTACTTACTTACTTACTTACGCGAA
S:      CTTACTTAC

T:  CGTGCCTACTTACTTACTTACTTACGCGAA
S:          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 :                                   G T A G C G G C G
```

completely ignored!

good suffix: 7