COMP261 Lecture 17

Marcus Frean

String Search 1 of 2

Victoria
UNIVERSITY OF WELLINGTON
Te Whare Wānanga
o te Ūpoko o te Ika a Māui
CAPITAL CITY UNIVERSITY

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

- Find the string "**vtewfvtxqwfczsrdzcaj**" in this text:

```
qwerxcvvtewfzxcfasfedrsadfsdacfasdrtvtewqwertcsvte
wfvtxqwfczsrdzfeceeaeszxcvtsafsersdxzcvtedfaevsadv
tewfvtxqwfczsvzxgvtasfvtcasrfvtewqtrwtravtewfxtrac
wrtrdtgfdvxvvsbdgfstqtretydfxvzccadawqeewtertgfvbd
vczfafsvtewfvtxqwfczsgfsdfdxvzvzvtvsdgfsgtfwt6fqwt
qwrcfxtvtewfwtqwfzvwqgtfvtqfwcxetwfazreqresdqxrdqc
fwqdxvgfewcvtwefxvtrfczrqesxqecaqrfzvtqwxvbwyegcbe
bcwtfexvtfwxcrqxeqdcqzrwdfvtwxefvctyvtewfwefxqtfxc
qcdzrqxesrzqxrqcwqtfxtewfcvwerygcvewytxvqewtcxzdcd
qwfxvtewfvtxqwfczsrdzcajwfcsxtqwefdvetwqfvxdtqfwvq
```

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?

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
```

- What is the cost?

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?

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

- anan#????????????????????????
  ananafg
  - what about now?
  - unsafe to jump straight 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.

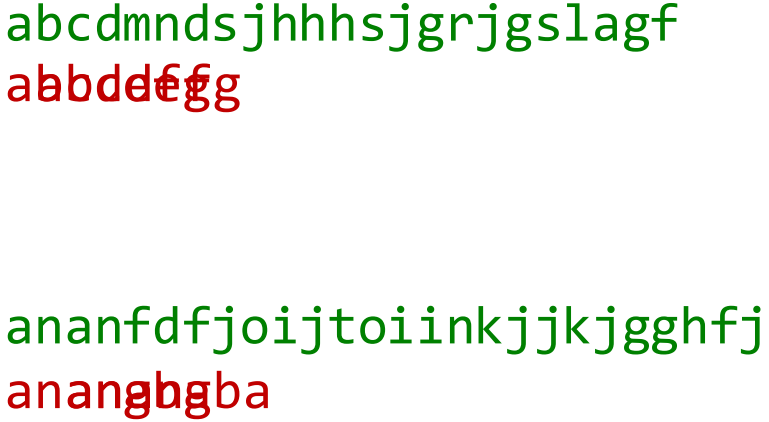
- never has to re-check a character

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 a "suffix ==prefix"?
      - No, skip these characters
        - » `M[s] = 0`
      - Yes, reuse, no need to recheck these characters
        - » `M[s]` is the length of the "reusable" suffix



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
```