Text Algorithms: String Matching

CS 2860: Algorithms and Complexity I

Magnus Wahlström, McCrea 113

Magnus.Wahlstrom@rhul.ac.uk

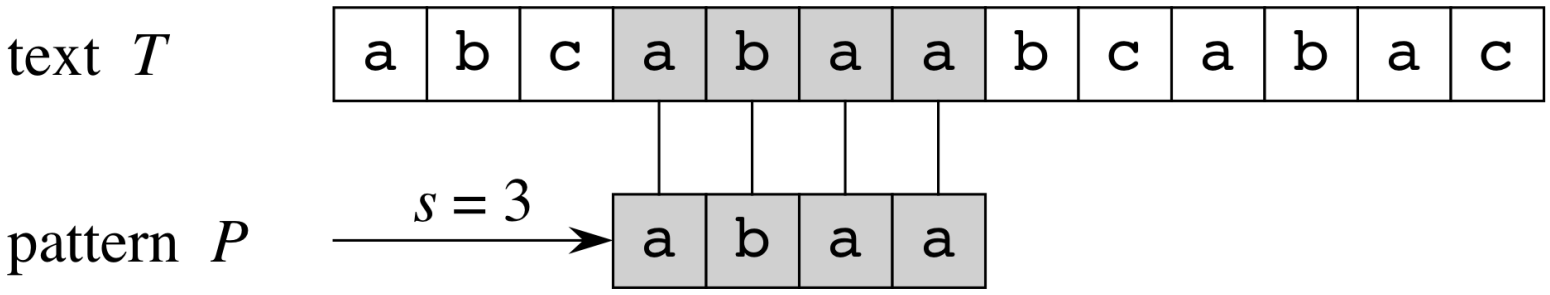
November 27, 2014

String Matching

String matching problem

- Situation: Have a long (n-character) text `T` and a shorter (m-character) pattern `P` that we are looking for in `T`
- Typical word processor problem: "Find the word `P` in the document `T`"
- Technical remark: Looking for `P` as substring in `T` (e.g., no "wildcards" in pattern)
  - Ex: "Something rotten in Denmark" contains pattern "ten in Den"
- Will see:
  - "Naïve" (worst-case slow) search algorithm
  - Later: Speedups and extensions

Terms and formalisation



- To us, the pattern `P` and text `T` are both strings: sequences of characters
- Formally, characters come from some alphabet Σ (say, normal English text symbols, e.g., one byte per character)
- Above: Pattern `P="abaa"` occurs in `T` at position 4 (or: with shift $s = 3$)
- Can also cover DNA sequences (alphabet $\Sigma = \{A, C, G, T\}$)

Naive string matching

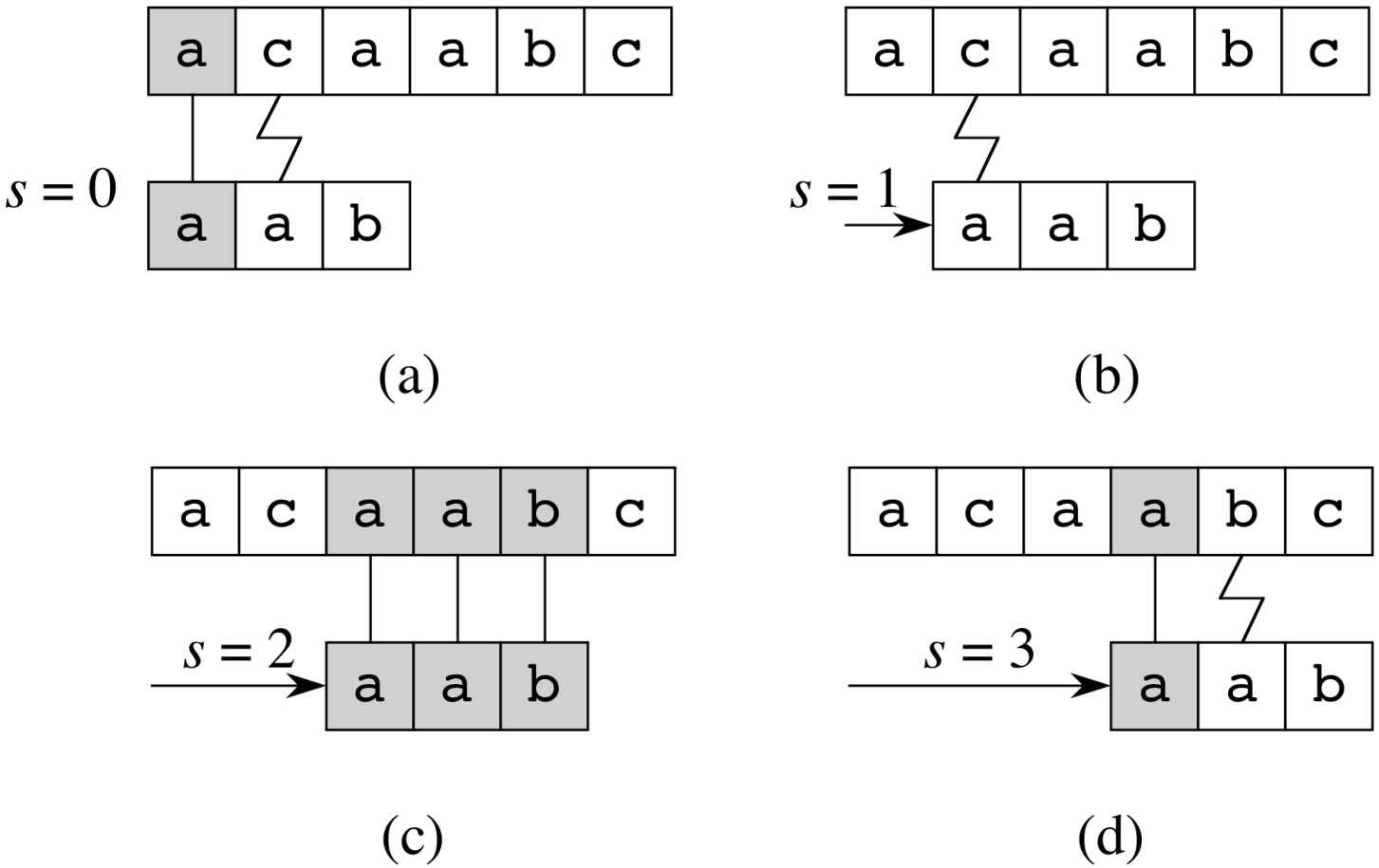
- Consider the simplest possible matching algorithm:
  1. For each possible shift s:
     1.1 If `P` occurs with shift s in `T`: Return `true`
  2. Otherwise, return `false`
- "For all possible shifts:" From `s=0` to `s=n-m` (why not `s=n`?)
- "Occurs" test: Does `P` occur at `s` in `T`?
  1. For `i=0` to `m-1`:
     1.1 If `T[s+i] != P[i]`: Pattern does not occur at `s`
  2. Otherwise, pattern occurs at `s`

Code

Ok, there are built-in functions for this, but for completeness:

```
boolean stringMatch(String text, String pattern) {
   int n=text.size(); int m=pattern.size();
   for (int pos=0; pos<=n-m; pos++) {
      boolean match=true;
      for (int i=0; i<m; i++) {
         if (text.charAt(pos+i) != pattern.charAt(i)) {
            match=false; break;
         }
      }
      if (match)
         return true; // or: return pos
   }
   return false; // or: return -1
}
```

Illustration



Analysis

- First of all: Naive search is usually quite good
- Usually, the inner loop fails early (e.g., first character wrong)
- But, for theoretical purposes, contrast the following cases:
  - Pattern `"abcd"` in any string
  - Pattern `P="aaab"` in string `T="aaaaaaaa"`
- Worst-case bound: (outer loops) $\times$ (inner loops) $=$ $(n-m) \times m = O((n-m)m) \approx O(nm)$.

String matching, overview

- Naive algorithm straight-forward, usually pretty good
- For further speedups, may preprocess string and/or pattern (if the data is big, e.g., DNA sequence)
  1. First, analyse pattern `P` (to learn how to search for it efficiently)
  2. Then, use this information in a faster search algorithm
- May also process the text to make it "friendlier" for searching

Rolling Hash Functions

String matching

- Let's understand the bottleneck of the simple matching algorithm:
  1. For `i=0` to `n-m`:
     1.1 Check if `pattern` occurs in `text` starting at position `i`
- Step 1 is performed $n - m + 1$ times, takes potentially $O(m)$ work each time
- Outside of course scope: Ways to fast-forward this search
- Will see: Replacement algorithm using hash functions (Rabin-Karp string matching)

Hash functions

- Recall hash functions: Kind of digital fingerprints or ID codes of objects
- For any objects `x` and `y`:
  - If `x.equals(y)`, then `x.hashCode() == y.hashCode()` (with certainty)
  - If `x` does not equal `y`, then very probably `x` and `y` have different hash codes
  - . . . unless `x` and `y` were chosen very unluckily, or maliciously
- Used in hash tables to "spread out" keys into slots
- We will use them to save on string comparisons

String matching via hashes

- Consider the following algorithm:
  1. Let `patternhash = pattern.hashCode()`
  2. For `i=0` to `n-m`:
     - 2.1 Compute hash code `texthash` of substring `text[i...i+m-1]`
     - 2.2 Only if `patternhash == texthash`, perform explicit string comparison (`pattern.equals(substring)`)
- Correctness:
  1. If pattern occurs, then `hashcode == texthash` and step 2 triggers
  2. Where pattern does not occur, may trigger step 2 anyway (rarely) – no harm
- Running time? (How quickly can we compute `texthash`?)

String hash functions

- For this, need inside knowledge of hash functions for strings
- Example: Java `String.hashCode()` function:
  - `pattern.hashCode() == pattern[0]`$\cdot 31^{m-1}$ `+ pattern[1]`$\cdot 31^{m-2}$ `+ ...+ pattern[m-1]`$\cdot 1$
- Questions:
  1. Good way of computing this?
  2. How does it speed up string matching?

Computing Java's hash functions

- Want to compute (string `s`, length m):
  - `s[0]`$\cdot 31^{m-1}$ `+ s[1]`$\cdot 31^{m-2}$ `+ ...+ s[m-1]`$\cdot$ `1`
- Consider the following (Horner's method):
  1. Start with `hash=0`
  2. For each character `c` (as integer) in string `s`:
     2.1 Let `hash = 31*hash + c`
- First steps:
  1. `s[0]`
  2. `31*s[0] + s[1]`
  3. `31*(31*s[0] + s[1]) + s[2]`
- In the end, `s[i]` has been multiplied by 31 exactly `m-i-1` times

Hash functions in Rabin-Karp

- So we know a good way to compute `String.hashCode()`
- What about computing hashcode of `text.substring(i,i+m)` for each i?
- Answer: Rolling hash functions
  - Want: value `text[i]`$\cdot 31^{m-1}$ `+ text[i+1]`$\cdot 31^{m-2}$ `+ ...+ text[i+m-1]`$\cdot 1$
  - Have: value `text[i-1]`$\cdot 31^{m-1}$`+ text[i]`$\cdot 31^{m-2}$ `+ ...+ text[i+m-2]`$\cdot 1$
  - Update step: `hash = (hash - text[i-1]`$\cdot 31^{m-1}$`) * 31 + text[i+m-1]`
- If we pre-compute 31^{m-1}, this is a constant-time update

Rabin-Karp string matching

- Final algorithm:
  1. Let `patternhash = pattern.hashCode()` and let `texthash` be the hash of `text[0...m-1]`
  2. For `i=0` to `n-m`:
     - 2.1 If `hashcode == texthash`, perform explicit string comparison (`pattern.equals(substring)`)
     - 2.2 Update `texthash` with new value `text[i+m]`, rolling out old value `text[i]`
- Searches for `pattern` in `text` in $O(n)$ time on average (under reasonable assumptions)
- Uses way to compute rolling hash function (hash codes of `text[i...i+m-1]` for all `i`) using $O(1)$-time update step

Rabin-Karp: Extensions

- Saw: Fast update step for computing hash code of substrings `text[i...i+m-1]`, to save on string comparisons – only call `String.equals` if hashes match
- Extension: Searching for many patterns in the text
  1. Assume that we are looking for patterns P1, P2, ..., Pt in the text, each of length m
  2. Put P1, ..., Pt in a hash table, using the above (rolling-friendly) function as hashcode
  3. Compute the rolling hashes for substrings of `text` as before, compare against hash table
  4. Perform explicit `String.equals` comparison whenever needed
- Example: Plagiarism test, look for one of many source sentences in student submission
- Performance: Average time $O(n)$