Advanced Algorithms in Bioinformatics (P4)

Sequence and Structure Analysis

Lecture / Tutorials / Software lab

Summer 2011, FU Berlin

David Weese, Sandro Andreotti

Web page at http://www.inf.fu-berlin.de/en/groups/abi/index.html

1.1 Aus der Studienordnung

Modul P4	**Fortgeschrittene Algorithmen in der Bioinformatik (10 LP)**
Zugangsvoraussetzungen (PO)	Grundkenntnisse der Algorithmischen Bioinformatik entsprechend den Lernzielen und -inhalten des Bachelor-Studiengangs Bioinformatik an der FU Berlin.
Qualifikationsziele und Inhalte (StO)	Dieses Forschungsmodul dient der Hinführung der Studierenden zu selbständiger wissenschaftlicher Arbeit in den an der FU Berlin vertretenen Forschungsschwerpunkten im Bereich der Algorithmischen Bioinformatik. Das Modul soll den aktuellen Stand der Forschung in diesem Gebiet repräsentieren. Es sollen vertieft Themen aus folgenden Gebieten behandelt werden: Verfahren für exakte und approximatives Suchen in Strings, Filter und Stringindices, sowie Alignments von RNA, DNA und Proteinsequenzen; Algorithmen zum Finden regulatorischer und codierender Signale in Gensequenzen; Verfahren zur Vorhersage molekularer Evolution auf Sequenzebene, Vergleichende Genomik; Algorithmen zur Auswertung von Gen- und Proteinexpressions-Experimenten; Struktur und Aufbau molekular-biologischer Datenbanken sowie Verbinden verschiedener Datenquellen
Lehr- und Lernformen/Formen der aktiven Teilnahme (StO)	Das Modul besteht aus einer Vorlesung mit Übung (3V + 1Ü). Diese werden entweder von einem Seminar oder einem Praktikum (2P) begleitet. Die aktive Teilnahme besteht dementsprechend aus der Teilnahme an den Übungen sowie einem Seminarvortrag oder Praktikumsberichten.
Modulprüfung (PO)	90-minütige Abschlussklausur
Zeitlicher Aufwand (StO)	300 Stunden
Häufigkeit des Angebots/Dauer des Moduls (StO)	Einmal im Jahr im Sommersemester. Die Veranstaltungen sind semesterbegleitend.

1.2 Vorläufige Themen

1. Exact and approximative string searching:
   Horspool, Wu-Manber, Shift-Or, Ukkonen, Myers, Banded Myers
2. Filtering algorithms:
   PEX, Schubfachprinzip, q-Gram Lemmata, QUASAR, gapped QUASAR, SWIFT
3. Index data structures:
   Constructing and Searching a Suffix Array, Enhanced Suffix Array (SA, LCP und Child Tab), FM-Index
4. Genome alignment:
   Match refinement, Chaining
5. RNA *de novo* folding:
   Nussinov, SCFGs, Zuker
6. RNA comparative folding & alignment:
   Covariance models

1.3 In der Praxis

- Erste zwei Drittel des Semesters (11. April bis 17. Juni):
  Vier Stunden Vorlesungen (Mo 14-16 in Arnimallee 14 SR FB (1.1.16), und Fr, 14-16, T9, SR006) und zwei Stunden Übung (Mi, 14-16, T9, SR006)
- Letztes Drittel (18. Juni bis 16. Juli):
  Praktikum. 6 Stunden, Mittwochs. (Arnimallee 6, R017 = Bioinformatik-Rechnerpool).

1.4 Anmeldung + Kommunikation

- Bitte tragen Sie sich (mit ihrem *FU*-Email-Account) in die Mailingliste zur Vorlesung und Übung ein:
  `https://lists.fu-berlin.de/listinfo/bioinf-P4-2011/`
- Aktuelle Nachrichten werden bevorzugt über die Mailingliste verbreitet, Downloads gibt es auf der Veranstaltungswebseite:
- Natürlich auch direkt nach der Vorlesung, in der Übung, in den Sprechstunden oder per e-mail.

1.5 Folien

- Folien: *lecture pool*–Konzept
- Die Folien
  - werden zeitnah online verfügbar gemacht. Eine Mitschrift des *Folien*inhaltes ist nicht nötig.
  - sind **KEIN** Skript. Hinzu kommen Erklärungen, Tafelanschrieb, Beispiele, Hinweise, Anekdoten, Overheadfolien, . . .
  - **gehen Sie in die Vorlesung** und machen Sie sich Notizen.
  - **lesen Sie ergänzende Literatur** (Liste jeweils am Anfang der entsprechenden Themen)

1.6 Praktikumsteil

- Praktische Themen aus dem Bereich der Vorlesung
- Kleingruppen (max. 3 Leute) bearbeiten ein Thema
- Regelmäßige Treffen Mittwochs im Bioinformatik–Pool.
- Programmierung dort oder woanders

1.7 Benotung

- Am Ende des Vorlesungsteils gibt es eine 90-minütige Klausur.
- Um die Veranstaltung mit der Note ausreichend oder besser abzuschließen, ist *aktive Teilnahme* erforderlich. Das bedeutet:
  - Regelmäßige aktive Teilnahme an den Übungen (Anwesenheits- und Bearbeitungspflicht, Vorrechnen)
  - Mindestens 50% der Punkte in den beiden Übungstests!
  - Aktive Teilnahme am Praktikum, d.h. ein Code Review und Bericht.

2 Fast exact string matching

We will discuss

- Horspool algorithm
- Wu-Manber algorithm

This exposition has been developed by C. Gröpl, G. Klau, and K. Reinert based on the following sources, which are all recommended reading:

- Flexible Pattern Matching in Strings, Navarro, Raffinot, 2002, pages 15ff.
- A nice overview of the plethora of exact string matching algorithms with animations implemented in java can be found under `http://www-igm.univ-mlv.fr/~lecroq/string`.

2.1 Thoughts about string matching

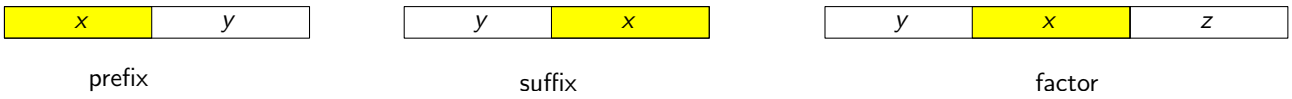
Let's start with the classical **exact string matching problem**:

> Find all occurrences of a given pattern $P = p_1, \ldots, p_m$ in a text $T = t_1, \ldots, t_n$, usually with $n \gg m$.

The algorithmic ideas of *exact* string matching are useful to know although in computational biology algorithms for *approximate* string matching or *indexing* methods are often in use. However, in online scenarios it is often not possible to precompute an index for finding exact matches.

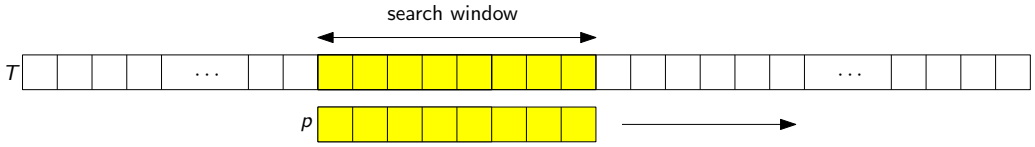
String matching is known for being amenable to approaches that range from the extremely theoretical to the extremely practical. In this lecture we will get to know two very practical exact string matching algorithms.

Some easy terminology: Given strings x, y, and z, we say that x is a *prefix* of xy, a *suffix* of yx, and a *factor* (:=substring) of yxz.

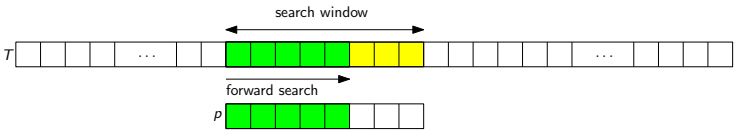


prefix suffix factor

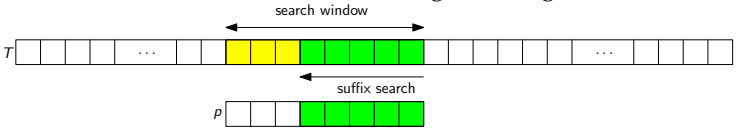
In general, string matching algorithms follow three basic approaches. In each a *search window* of the size of the pattern is slid from left to right along the text and the pattern is searched within the window. The algorithms differ in the way the window is shifted.



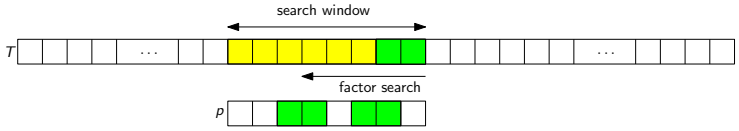
1. *Prefix searching*. For each position of the window we search the longest prefix of the window that is also a prefix of the pattern.



2. *Suffix searching*. The search is conducted backwards along the search window. On average this can avoid to read some characters of the text and leads to sublinear average case algorithms.



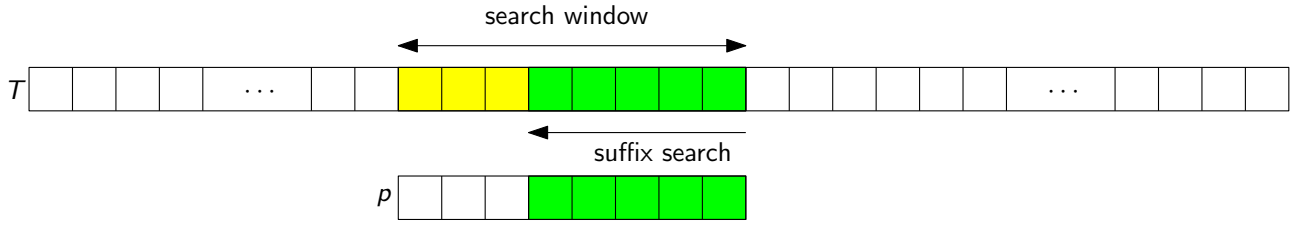
3. *Factor searching*. The search is done backwards in the search window, looking for the longest suffix of the window that is also a factor of the pattern.



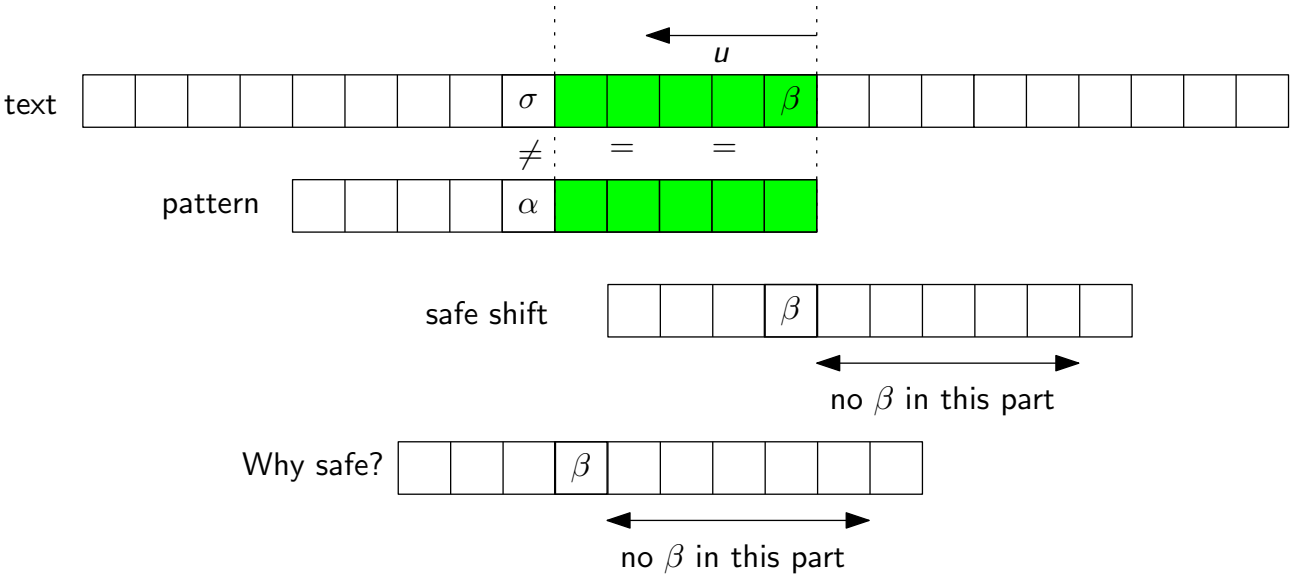
2.2 Idea of the Horspool algorithm

As noted above, in the suffix based approaches we match the characters from the back of the search window. Whenever we find a mismatch we can shift the window in a safe way, that means without missing an occurrence of the pattern.

We present the idea of the *Horspool* algorithm, which is a simplification of the *Boyer-Moore* algorithm.



For each position of the search window we compare the last character β with the last character of the pattern. If they match we verify until we find the pattern or fail on the text character σ. Then we simply shift the window according to the next occurrence of β in the pattern.



2.3 Horspool pseudocode

Input: text T of length n and pattern p of length m
Output: all occurences of p in T

Preprocessing:
for $c \in \Sigma$ **do** $d[c] = m$;
for $j \in 1 \ldots m-1$ **do** $d[p_j] = m - j$

Searching:
$pos = 0$;
while $pos \leq n - m$ **do**
    $j = m$;

while $j > 0 \wedge t_{pos+j} = p_j$ **do** $j{-}{-}$;
if $j = 0$ **then output** "p occurs at position $pos + 1$"
$pos = pos + d[t_{pos+m}]$;

We notice two things:

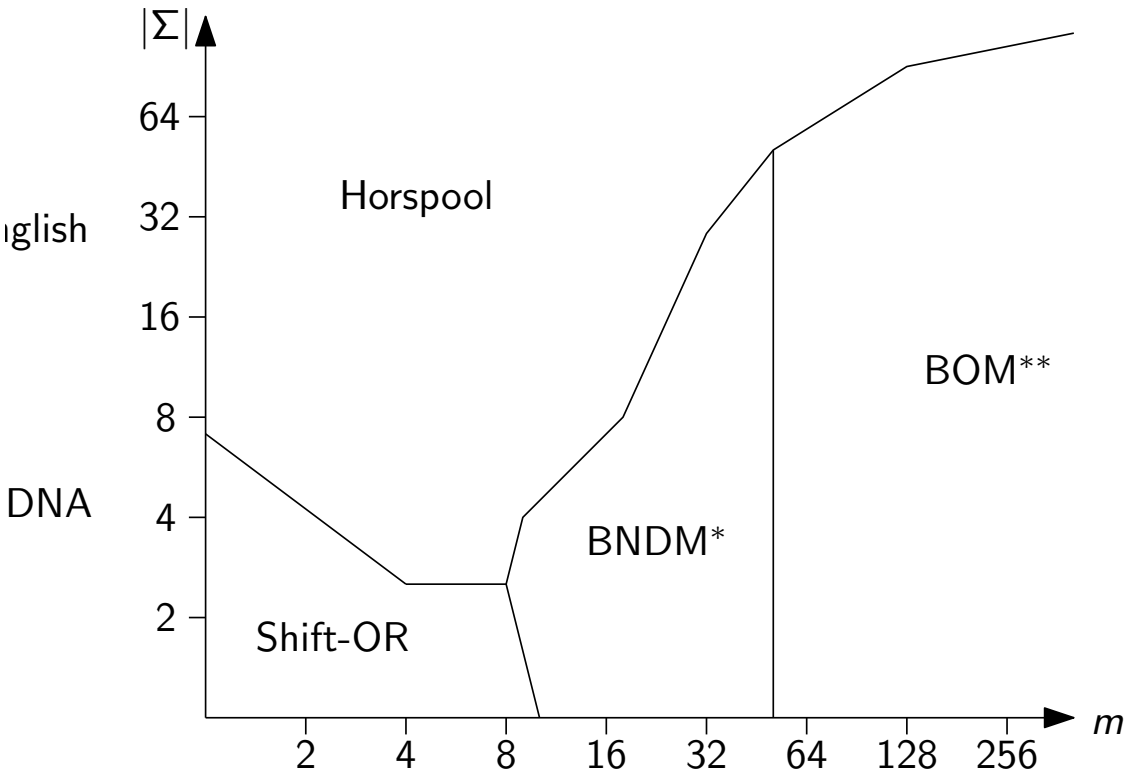
1. The verification could also be done forward. Many implementations use built-in memory comparison instructions of the machines (i. e. `memcmp`). (The java applet also compares forward.)
2. The main loop can be "unrolled", which means that we can first shift the search window until its last character matches the last character of the pattern and then perform the verification.

2.4 Horspool example

```
pattern: announce    text: cpmxannualxconferencexannounce

  bmBc:
 a c e f l m n o p r u x
 7 1 8 8 8 8 2 4 8 8 3 8

attempt 1:
cpmxannualxconferencexannounce
.......e
Shift by 3 (bmBc[u])

attempt 2:
cpmxannualxconferencexannounce
   .......e
Shift by 8 (bmBc[x])

attempt 3:
cpmxannualxconferencexannounce
           .......e
Shift by 2 (bmBc[n])

attempt 4:
cpmxannualxconferencexannounce
             a......E
Shift by 8 (bmBc[e])

attempt 5:
cpmxannualxconferencexannounce
                        .......e
Shift by 1 (bmBc[c])

attempt 6:
cpmxannualxconferencexannounce
                          ANNOUNCE
Shift by 8 (bmBc[e])


cpmxannualxconferencexANNOUNCE

String  length: 30
Pattern length: 8
Attempts: 6
Character comparisons: 14
```

2.5 Experimental Map

(Gonzalo Navarro & Mathieu Raffinot, 2002)



ondeterministic DAWG Matching algorithm, factor-based, not covered in this lecture
Oracle Matching, factor-based, not covered in this lecture

2.6 Thoughts about multiple exact string matching

The **multiple exact string matching problem** is:

Find all occurrences of a given set of r patterns $P = \{p^1, \ldots, p^r\}$ in a text $T = t_1, \ldots, t_n$, usually with $n \gg m_i$. Each p^i is a string $p^i = p^i_1, \ldots, p^i_{m_i}$. We denote with $|P|$ the total length of all patterns, i. e. $|P| = \sum_{i=1}^{r} \left|p^i\right| = \sum_{i=1}^{r} m_i$.

As with single string matching we have three basic approaches:

1. *Prefix searching*. We read each character of the string with an automaton built on the set P. For each position in the text we compute the longest prefix of the text that is also a prefix of one of the patterns.
2. *Suffix searching*. A positions *pos* is slid along the text, from which we search backward for a suffix of any of the strings.
3. *Factor searching*. A position *pos* is slid along the text from which we read backwards a factor of some prefix of size *lmin* of the strings in P.

2.7 Suffix based approaches

For single string matching, the suffix based approaches are usually faster than the prefix based approaches, hence it is natural to extend them to sets of patterns. The first algorithm with sublinear expected running time was that of *Commentz-Walter* in 1979. It is a direct extension of the *Boyer-Moore* algorithm. The search window is verfied from right to left using a trie for the set of reversed patterns, $P^{\text{rev}} = \left\{(p^1)^{\text{rev}}, \ldots, (p^r)^{\text{rev}}\right\}$. Again there are three rules to determine a safe shift.

The *Horspool* algorithm also has a straightforward extension to multiple patterns. But it is much less powerful matching than for a single pattern, because the probability to find any given character in one of the strings gets higher and higher with the number of strings. A stronger extension is the *Wu-Manber* algorithm which is in fact superior to other algorithms for most settings.

Do you see why the performace of the Horspool algorithm deteriorates quickly as the number of patterns increases?

2.8 Wu-Manber algorithm

Next we introduce a suffix-search based multi-pattern search algorithm, the *Wu-Manber* algorithm. Recall that we want to search simultaneously for a set of r strings $P = \{p^1, p^2, \ldots, p^r\}$ where each P^i is a string $p^i = p^i_1, p^i_2 \ldots p^i_{m_i}$. Let *lmin* be the minimum length of a pattern in P and *lmax* be the maximum length. As usual we search in a text $T = t_1 \ldots t_n$.

The key idea of Wu and Manber is to use *blocks* of characters of length B to avoid the weakness of the Horspool algorithm.

For each string of length B appearing at the end of the window, the algorithm "knows" a safe shift (after some preprocessing).

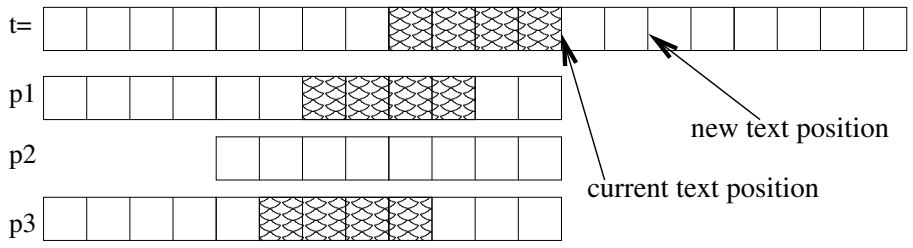
Instead of using a table of size $|\Sigma|^B$ each possible block is assigned a *hash value* which is used to store the blocks in hash tables. Note that $|\Sigma|^B$ can be quite a large number, and the distibution of blocks is usually very unevenly.

You can ignore the hash function for a moment to catch up the main idea (i.e., take the identity as a hash function). (Don't forget to think about it later, though.)

The algorithm uses two tables *SHIFT* and *HASH*.

SHIFT(j) contains a *safe* shift, that means it contains the minimum of the shifts of the blocks *Bl* such that $j = h_1(Bl)$. More precisely:

- If a block *Bl* does not appear in any string in P, we can safely shift $lmin - B + 1$ characters to the right. This is the default value of the table.
- If *Bl* appears in one of the strings of P, we find its rightmost occurrence in a string p^i, let j be the position where it ends, and set $SHIFT(h_1(Bl))$ to $m_i - j$.



SHIFT[t(i−B+1),...,t_(i)]=min{2,3,5}

To compute the *SHIFT* table consider separately each $p^i = p^i_1 \ldots p^i_{m_i}$. For each block $Bl = p^i_{j-B+1} \ldots p^i_j$, we find its corresponding hash value $h_1(Bl)$ and store in $SHIFT(h_1(Bl))$ the minimum of the previous value and $m_i - j$.

During the search phase we can shift the search positions along the text as long as the *SHIFT* value is positive. When the shift is zero, the text to the left of the search position might be one of the pattern strings.

The entry j in table HASH contains the indices of all patterns that end with a block *Bl* with $h_2(Bl) = j$. This table is used later for the verification phase.

To access the possible strings, $HASH(h_2(Bl))$ contains a list of all pattern strings whose last block is hashed to $h_2(Bl)$. In the original paper Wu and Manber chose $h_1 = h_2$ to save a second hash value computation.

2.9 Wu-Manber pseudocode

```
1 WuManber(P = {p^1, p^2, ..., p^r}, T = t_1 t_2 ... t_n)
2 // Preprocessing
3 Compute a suitable value of B    (e.g. B = log_|Σ|(2 · lmin · r));
4 Construct Hash tables SHIFT and HASH;
5 // Searching
6 pos = lmin;
```

```
7  while pos ≤ n do
8        i = h1(t_{pos−B+1} ... t_{pos});
9        if SHIFT[i] = 0
10         then
11               list = HASH[h2(t_{pos−B+1} ... t_{pos})];
12               Verify all patterns in list against the text;
13               pos++;
14         else
15               pos = pos + SHIFT[i];
16       fi
17 od
```

2.10 Wu-Manber example

Assume we search for $P = \{announce, annual, annually\}$ in the text $T = CPM\_annual\_conference\_announce$. Assume we choose $B = 2$ and a suitable hashfunction (exercise) and assume we are given the following tables:

	ll	no,ou	an	un,nc	ua,al	ly	nn,nu	ce	*
SHIFT(BL)=	1	3	4	1	0	0	2	0	5

HASH(BL) =	ce,ly	al	*
	3,1	2	nil

Then the algorithm proceeds as follows:

- CPM_annual_conference_announce
  SHIFT[an] = 4.
- CPM_annual_conference_announce
  SHIFT[al] = 0. List = HASH[al] = {2}.
  Compare p^2 against the string and mark its occurrence. Shift search positions by 1.
- CPM_annual_conference_announce
  SHIFT[l_] = 5.
- CPM_annual_conference_announce
  SHIFT[fe] = 5.
- CPM_annual_conference_announce
  SHIFT[ce] = 0. List = HASH[ce] = {3, 1}.
  Compare p^1 and p^3 against the text. No string matches. Shift by one.
- CPM_annual_conference_announce
  SHIFT[e_] = 5.
- CPM_annual_conference_announce
  SHIFT[ou] = 3.
- CPM_annual_conference_announce
  SHIFT[ce] = 0. List = HASH[ce] = {3, 1}.
  Compare p^1 and p^3 against the text. Test succeeds for p^1. Mark its occurrence.