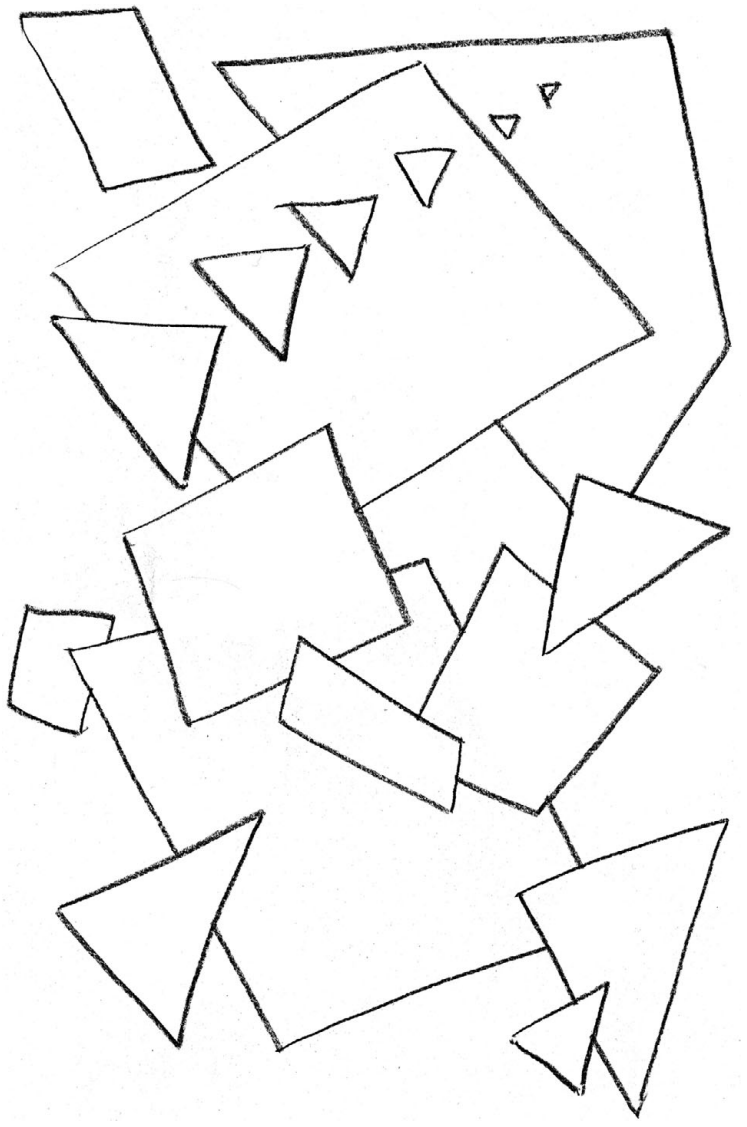
3 Pattern Matching



https://doi.org/10.1017/9781108835831.004 Published online by Cambridge University Press

19 Border Table

The border table, as well as the prefix table in Problem 22, are basic tools for building efficient algorithms on words. They are used mostly for searching texts online for various types of given patterns.

The ***border table*** of a non-empty word x is defined on the lengths ℓ, $\ell = 0, \ldots, |x|$, of its prefixes both by $border[0] = -1$ and, for $\ell > 0$, by $border[\ell] = |Border(x[0\,..\,\ell - 1])|$. Here is the border table of the word `abaababaaba`:

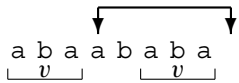
i		0	1	2	3	4	5	6	7	8	9	10
$x[i]$		a	b	a	a	b	a	b	a	a	b	a
ℓ	0	1	2	3	4	5	6	7	8	9	10	11
$border[\ell]$	-1	0	0	1	1	2	3	2	3	4	5	6

Question. Prove that Algorithm BORDERS correctly computes the border table of a non-empty word x and executes a maximum of $2|x| - 3$ letter comparisons when $|x| > 1$.

BORDERS(x non-empty word)

```
1  border[0] ← −1
2  for i ← 0 to |x| − 1 do
3      ℓ ← border[i]
4      while ℓ ≥ 0 and x[ℓ] ≠ x[i] do
5          ℓ ← border[ℓ]
6      border[i + 1] ← ℓ + 1
7  return border
```

Example. Let us consider the prefix u = `abaababa` of the above word. Its border is v = `aba` of length 3 = $border[8]$. The next letter `a` extends the border, that is, $Border(u\mathtt{a}) = v\mathtt{a}$.



If the next letter c is not `a`, the border of uc is the border of vc, which sketches the proof of the recurrence relation, for u a word and c a letter:

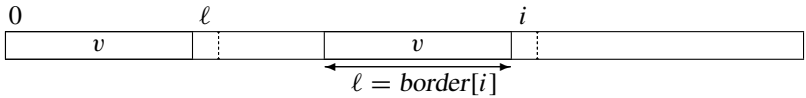
$$Border(uc) = \begin{cases} Border(u)c & \text{if } Border(u)c \text{ is a prefix of } u, \\ Border(Border(u)c) & \text{otherwise.} \end{cases}$$

https://doi.org/10.1017/9781108835831.004 Published online by Cambridge University Press

Question. Show how to detect the non-primitive prefixes of a word using its border table.

Solution

The proof of correctness relies on the above recurrence relation. It can be restated by saying that the second longest border of a word u, if it exists, is the border of its border.



Correctness of Algorithm Borders. First note that *border*[0] is set up correctly.

Instructions in the for loop, lines 3–6, compute *border*$[i+1]$ from all the other values *border*$[i']$, $i' = 0, \ldots, i$. During the execution of the while loop, ℓ runs through lengths of the borders of $x[0\,..\,i-1]$ from the longest to the shortest.

If the while stops and $\ell = -1$, no border can be extended by the letter $x[i]$; then the next border is empty, that is, *border*$[i+1] = 0$, as done. Otherwise, the loop stops the first time a match is found providing the longest border of $x[0\,..\,i-1]$, of length ℓ, extensible by letter $x[i]$. Then *border*$[i+1] = \ell + 1$, as done again. This ends the proof.

Running time of Algorithm Borders. Note it is sufficient to establish the bound to prove the linear running time. To do it we show that the value of $2i - \ell$ increases by at least 1 after each letter comparison. Indeed, i and ℓ are both incremented after a positive comparison and ℓ is decreased by at least 1 with i unchanged after a negative comparison.

When $|x| > 1$, the value of $2i - \ell$ runs from 2, for the first comparison with $i = 1$ and $\ell = 0$, to at most $2|x|-2$ during the last comparison with $i = |x|-1$ and $\ell \geq 0$. The overall number of comparisons is thus bounded by $2|x| - 3$, as stated. The bound of $2|x| - 3$ is reached by any string x of the form $a^{|x|-1}b$, where a and b are different letters.

Non-primitive prefixes. By definition, a non-primitive word u is of the form w^k for some non-empty word w and an integer $k > 1$. It means $|u| = k|w|$; that is, the period $|w|$ of u divides its length $|u|$. Therefore the detection of a non-primitive prefix of length i of x from the border table of x can be done by checking if the exponent $i/(i - border[i])$ of the prefix is an integer greater than 1.

https://doi.org/10.1017/9781108835831.004 Published online by Cambridge University Press

Notes

The use of border tables for matching words is a classical topic in textbooks like [74, 96, 134, 194, 228]. The initial design is by Morris and Pratt (see [162]).

Since $|u| - border[|u|]$ is the smallest period of u, the border table of a word can be transformed into the table of periods of its prefixes. A striking solution for computing this latter table for a Lyndon word is shown in Problem 42.

20 Shortest Covers

The notion of cover tries to capture the regularity of a word. It goes beyond the possible periodicity of the word by considering an a priori shorter factor that covers the whole word. Period words are specific covers that occur consecutively in the word while general covers may have overlapping occurrences. In that sense the notion of cover generalises the notion of period. More accurately, a cover u of a word x is a border of x whose consecutive occurrence positions are at maximum distance $|u|$.

Example. The word $u = \texttt{aba}$ is a cover of the word $x = \texttt{abaababa}$. Indeed, the sorted list of starting positions of occurrences of u in x is $pos(\texttt{aba}, \texttt{abaababa}) = (0, 3, 5)$ and the maximum distance consecutive positions of the list is $MaxGap(\{0, 3, 5\}) = 3 \leq |u|$. The word u is the shortest cover of x.

The shortest cover of abaababaaaba is the whole word, which is always a trivial cover of itself. When this condition holds the word is said to be *superprimitive*. It is also primitive.

The ***cover table***, denoted by *cover*, associated with a word x is defined on length ℓ of its prefixes as follows: $cover[\ell]$ is the smallest length of covers of $x[0\,..\,\ell - 1]$. Here is the cover table of the word abababaaba.

i		0	1	2	3	4	5	6	7	8	9
$x[i]$		a	b	a	b	a	b	a	a	b	a
ℓ	0	1	2	3	4	5	6	7	8	9	10
$cover[\ell]$	0	1	2	3	2	3	2	3	8	9	3

https://doi.org/10.1017/9781108835831.004 Published online by Cambridge University Press

Question. Design a linear-time algorithm computing the shortest cover of each prefix of a word.

[**Hint:** Use the border table of the word.]

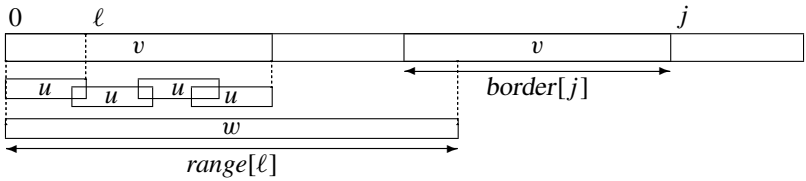
Solution

The present solution, an online algorithm to compute the cover table of a word x, relies on a key observation. (The border table of a word is treated in Problem 19.)

Observation. The only candidate for a non-trivial shortest cover of $x[0\,..\,j-1]$ is the shortest cover $u = x[0\,..\,\ell-1]$ of $v = x[0\,..\,border[j]-1]$, which is the (longest) border of $x[0\,..\,j-1]$ (see picture). This is because any non-trivial cover of $x[0\,..\,j-1]$ is a cover, possibly trivial, of its border.

In addition, the algorithm makes a crucial use of a supplementary table *range*: $range[\ell]$ is the length of the longest prefix of $x[0\,..\,j-1]$ covered by $x[0\,..\,\ell-1]$ (prefix v is covered by u on the picture). The next observation explains the role of the table *range*.

Observation. If $u = x[0\,..\,\ell-1]$ is a cover of the border of $x[0\,..\,j-1]$ and $range[\ell]$ is as large as the period of $x[0\,..\,j-1]$, u is a cover of $x[0\,..\,j-1]$.



Algorithm ShortestCovers implements the observations.

```
ShortestCovers(x non-empty word)
 1  border ← Borders(x)
 2  for j ← 0 to |x| do
 3       (cover[j], range[j]) ← (j, j)
 4  for j ← 2 to |x| do
 5       ℓ ← cover[border[j]]
 6       if ℓ > 0 and (range[ℓ] ≥ j − border[j]) then
 7            (cover[j], range[ℓ]) ← (ℓ, j)
 8  return cover
```

https://doi.org/10.1017/9781108835831.004 Published online by Cambridge University Press

After a run of Algorithm SHORTESTCOVERS on the above example ababababaaba, the tables get their final values:

j	0	1	2	3	4	5	6	7	8	9	10
$border[j]$	-1	0	0	1	2	3	4	5	1	2	3
$range[j]$	0	1	6	10	4	5	6	7	8	9	10
$cover[j]$	0	1	2	3	2	3	2	3	8	9	3

Note that super-primitive prefixes are those whose length j satisfies $cover[j] = j$.

Following the computation of the table *border*, the instruction at lines 3 initialises trivially tables *cover* and *range*. Instructions at lines 4–7 compute $cover[j]$ and update $range$. The condition ($range[\ell] \geq j - border[j]$) at line 6 checks, according to the second observation, whether ℓ is actually the length of the shortest cover of $x[0\,..\,j-1]$. This completes the proof of correctness.

Since Algorithm BORDERS runs in linear time, this is also the case for Algorithm SHORTESTCOVERS.

Notes

The present algorithm was designed by Breslauer in [43] to test the super-primitivity of words.

21 Short Borders

The problem deals with a special type of border table of a word. It is adapted to search texts for Zimin patterns containing word variables (see Problem 43). It shows the notion of border table is powerful when tuned for searching online for various types of patterns.

A border of a non-empty word x is any word that is both a proper prefix and a suffix of it. A border is said to be *short* if its length is smaller than $|x|/2$. The notations $Border(x)$ and $ShortBorder(x)$ stand for the longest border of x and for its longest short border, respectively. Any of these borders can be the empty word.

For example, borders of $x = \texttt{abababab}$ are ε, ab, abab and abababab. Only ε and ab are short, $Border(x) = \texttt{abababab}$ and $ShortBorder(x) = \texttt{ab}$.

https://doi.org/10.1017/9781108835831.004 Published online by Cambridge University Press

In some algorithms the notion of a short border is more appropriate when known for all the prefixes of a word. Let *shbord* denote the ***short-border table*** of prefixes of a non-empty word x. It is defined on prefix lengths ℓ, $0 < \ell \leq |x|$, by

$$shbord[\ell] = |ShortBorder(x[0\,..\,\ell-1])|$$

(by convention $shbord[0] = -1$). Below are the tables for the word `abaababaaba`. They differ at positions $\ell = 6, 10, 11$.

i		0	1	2	3	4	5	6	7	8	9	10
$x[i]$		a	b	a	a	b	a	b	a	a	b	a
ℓ	0	1	2	3	4	5	6	7	8	9	10	11
$border[\ell]$	−1	0	0	1	1	2	3	2	3	4	5	6
$shbord[\ell]$	−1	0	0	1	1	2	1	2	3	4	2	3

Question. Design a linear-time algorithm computing the table *shbord* of a non-empty word.

Solution

A straightforward solution would be to compute the table *shbord* from the table *border* of the word without looking at the word itself. But this is likely to yield a quadratic execution time on examples like $\mathtt{a}^k$ or $(\mathtt{ab})^k$.

Instead, Algorithm ShortBorders, which still uses the table of borders, is a modification of Algorithm Borders that computes this table (see Problem 19). Then it also runs in linear time. It tries to enlarge the previous short border and when the extension is too long uses the table of borders to switch to a shorter border.

```
ShortBorders(x non-empty word)
 1  border ← Borders(x)
 2  shbord[0] ← −1
 3  for i ← 0 to |x| − 1 do
 4      ℓ ← shbord[i]
 5      while (ℓ ≥ 0 and x[ℓ] ≠ x[i]) or (2ℓ + 1 ≥ i) do
 6          ℓ ← border[ℓ]
 7      shbord[i + 1] ← ℓ + 1
 8  return shbord
```

https://doi.org/10.1017/9781108835831.004 Published online by Cambridge University Press

The correctness of Algorithm SHORTBORDERS follows from the fact that the next short border is an extension of the previous short border of $u = x[0\,..\,i-1]$ by the single symbol $x[i]$, or is an extension of a shorter border of u.

Since the number of executions of the instruction at line 6 is bounded by the number of increments of ℓ, the overall running time is $O(|x|)$, as expected.

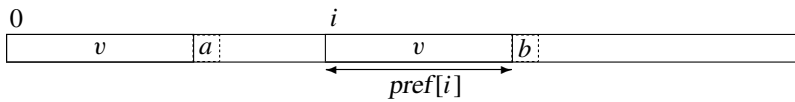
Notes

The motivation to introduce and compute the table *shbord* is its essential implication in the computation of Zimin types of words and their crucial use in algorithms fast searching for a given Zimin pattern (containing variables) in words (without variables) (Problem 43).

22 Prefix Table

The prefix table, like the border table of Problem 19, is a basic tool for building efficient algorithms on words. It is used mostly when searching texts for various types of given patterns.

Let x be a non-empty string. The ***prefix table*** of x is defined on its positions i, $i = 0, \ldots, |x| - 1$, by: $pref[i]$ is the length of the longest prefix of x starting at position i. Obviously $pref[0] = |x|$.



Here is the prefix table of the word `abaababaaba`:

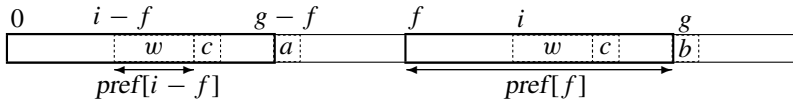
i	0	1	2	3	4	5	6	7	8	9	10
$x[i]$	a	b	a	a	b	a	b	a	a	b	a
$pref[i]$	11	0	1	3	0	6	0	1	3	0	1

Question. Show that Algorithm PREFIXES computes in linear time the prefix table of its input word x and executes no more than $2|x| - 2$ letter comparisons.

https://doi.org/10.1017/9781108835831.004 Published online by Cambridge University Press

```
PREFIXES(x non-empty word)
 1  pref[0] ← |x|
 2  g ← 0
 3  for i ← 1 to |x| − 1 do
 4      if i < g and pref[i − f] ≠ g − i then
 5          pref[i] ← min{pref[i − f], g − i}
 6      else (f, g) ← (i, max{g, i})
 7          while g < |x| and x[g] = x[g − f] do
 8              g ← g + 1
 9          pref[f] ← g − f
10  return pref
```

The key idea in Algorithm PREFIXES that computes the table sequentially from left to right is to benefit from what has been computed before the current position.



When $v = x[f \,..\, g-1]$ is a prefix of x and position i is between f and g (see picture), the first step for computing $pref[i]$ is to check whether or not its value can be deduced from the work done on the prefix occurrence of v, that is, at position $i - f$ on x. This saves enough work to get a linear-time algorithm.

Solution

Correctness of PREFIXES. Let us first clarify the role of variables f and g. At some point during a run of the algorithm, the position g is the farthest position to the right where a (negative) letter comparison happened. More accurately, for a given position i, $g = \max\{j + pref[j] : 0 < j < i\}$. And the associated position f satisfies $f + pref[f] = g$.

During the first pass in the for loop, f and g are set as well as is $pref[i] = pref[f]$ in accordance with their definitions by mere letter comparisons, which gives the invariant of the loop.

To show the invariant is maintained during other passes in the for loop, we examine what instructions at lines 4–9 do.

If $i < g$ and $pref[i - f] < g - i$ it is clear that $pref[i] = pref[i - f]$; see the above picture. If $i < g$ and $pref[i - f] > g - i$ the longest prefix of x starting at position $i - f$ is of the form $v'x[g - f]v''$, where v' is a suffix of

https://doi.org/10.1017/9781108835831.004 Published online by Cambridge University Press

$x[i-f\,..\,g-f-1] = x[f\,..\,g-1]$. Then, since $x[g] \neq x[g-f]$, v' is the longest prefix of x starting at i, that is, $pref[i] = g-i$. Therefore, when $i < g$ and $pref[i-f] \neq g-i$, the instruction at line 5 sets correctly the value of $pref[i]$ without changing the invariant.

When the condition at line 4 does not hold, as in the first pass, the value of $pref[i]$ is set correctly by letter comparisons and the invariant is satisfied after that pass. This ends the proof of correctness.

Running time of PREFIXES. The running time depends essentially on the number of letter comparisons at line 7. There is at most one negative comparison for each value of i because this stops the execution of the while loop, that is, at most $|x|-1$. There is at most one positive comparison for each value of g because it increases the value of g that never decreases, that is, at most $|x|-1$ again. The total number of letter comparisons is thus no more that $2|x|-2$, which shows that the overall running time is $O(|x|)$.

Notes

Prefix tables, as well as border tables in Problem 19, are basic notions on words to design algorithms on texts, presented sometimes implicitly in textbooks like [74, 96, 134, 194, 228]. Algorithm PREFIXES is called the Z algorithm in [134, page 9].

23 Border Table to the Maximal Suffix

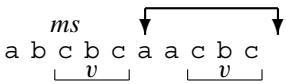
The maximal suffix of a word, its alphabetically greatest suffix, helps designing optimal text searches and periodicity tests. The problem introduces a computation of it based on the border table algorithm.

Question. Show that the following version of the border table computation correctly returns the starting position on the input word of its maximal suffix and that it runs in linear time.

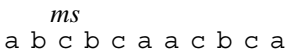
https://doi.org/10.1017/9781108835831.004 Published online by Cambridge University Press

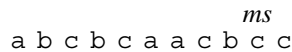
```
MaxSuffixPos(x non-empty word)
 1  ms ← 0
 2  border[0] ← −1
 3  for i ← 0 to |x| − 1 do
 4      ℓ ← border[i − ms]
 5      while ℓ ≥ 0 and x[ms + ℓ] ≠ x[i] do
 6          if x[ms + ℓ] < x[i] then
 7              ms ← i − ℓ
 8          ℓ ← border[ℓ]
 9      border[i − ms + 1] ← ℓ + 1
10  return ms
```

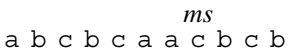
Example. The word $x = \texttt{abcbcacbc}$ has ***maximal suffix*** MaxSuffix$(x) = \texttt{cbcacbc}$ whose longest border is $v = \texttt{cbc}$. A next letter is to be compared with the letter following the prefix cbc of the maximal suffix.

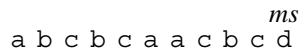


The following pictures show how the maximal suffix evolves according to the appended letter a, b, c or d. Note that if the periodicity of the maximal suffix changes, the maximal suffix is of the form va where v is a border of the initial maximal suffix and a is the new letter.







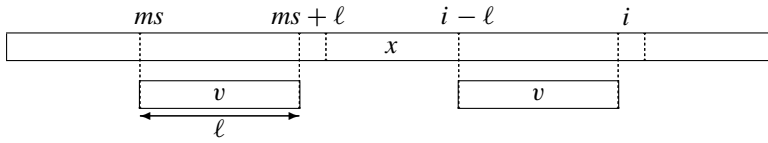


Question. Show how to find the alphabetically largest conjugate (rotation) of a word.

Solution

The online algorithm is based on the above remark on the border of the previous maximal suffix. The role of instructions inside the for loop is simultaneously to update the starting position of the maximal suffix and to compute the length of its border. Note that without instruction at lines 6–7, the variable *ms* remains null and the algorithm computes the table *border* of the whole word x.

https://doi.org/10.1017/9781108835831.004 Published online by Cambridge University Press



Correctness of MaxSuffixPos. The proof is done by showing the following invariant holds after executing instructions inside the for loop, lines 4–9: ms is the starting position of the maximal suffix of $x[0\,..\,i-1]$ and $border[i-ms]$ is the border length of this suffix of length $i-ms$.

After the initial instructions and the first pass in the loop for $i=0$, we get $ms=0$ and $border[1]=0$, the correct result for the word $x[0]$.

By induction, other passes start by setting ℓ to the border length of $x[ms\,..\,i-1]$. The fact that the border length of $x[ms\,..\,i]$ is correctly computed follows the same argument as in the construction of the border tables in Problem 19 about the iteration of *border* at line 8. In short, when the while loop stops ms was updated at line 7 each time the border v of the current maximal suffix satisfies $vx[ms+\ell] < ux[i]$, which effectively locates the maximal suffix of $x[ms\,..\,i]$.

Running time of MaxSuffixPos. This is the same running time as for the computation of the border table (see Problem 19), that is, $O(|x|)$, since instruction at lines 6–7 takes constant time.

Largest conjugate. Let $ms = \text{MaxSuffixPos}(y)$, where $y = xx$. Then the largest conjugate of x is $y[ms\,..\,ms+|x|-1]$. Modular arithmetic on indices can be used to avoid duplicating x.

Notes

The above algorithm is adapted from the computation of the smallest rotation of a circular word by Booth [39].

Two other solutions for the computation of the maximal suffix of a word are shown in Problems 38 and 40. They also run in linear time but with the advantage of requiring only constant extra memory space. The second additionally reports the period of the maximal suffix.

https://doi.org/10.1017/9781108835831.004 Published online by Cambridge University Press

24 Periodicity Test

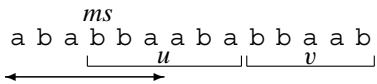
The detection of periodicities in words is an essential question to cope with when designing text searching or text compression methods. The goal is to test the periodicity of a word in a space-economical way.

A word x is said to be ***periodic*** if its (smallest) period is no more than half its length, $per(x) \leq |x|/2$. Equivalently, x is periodic if it is of the form $u^k v$ for two words u and v and an integer k, u non-empty, v a proper prefix of u and $k \geq 2$.

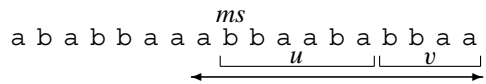
Checking the property is straightforward from the border or prefix tables of x. The goal here is to do it still in linear time but with only constant extra space (in addition to input). The idea is to use the function MaxSuffixPos of Problem 40 that gives the position and period of the maximal suffix of x.

The examples below illustrate variations of the period of a word according to its maximal suffix. The maximal suffix starts at position ms and is in the form $u^k v$ (here $k = 1$), where $u = x[ms \,..\, ms + p - 1]$, p is its period and v is a proper prefix of u.

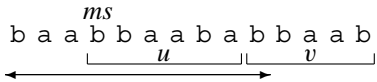
First, x = `ababbaababbaab`. The prefix `aba` of x preceding the maximal suffix uv is a suffix of u. Then x is periodic with period 6.



Second, x = `ababbaaabbaababbaa`. The prefix of x preceding uv is longer than u. The period of x is $11 > |uv|$ and x is not periodic.



Third, x = `baabbaababbaab`. The prefix of x preceding uv is shorter than u but not a suffix of it. The period of x is $10 > |x| - |v|$ and x is not periodic.



Question. Show that the periodicity of a word x can be tested with less than $|x|/2$ letter comparisons if the starting position and the period of its maximal suffix are given.

https://doi.org/10.1017/9781108835831.004 Published online by Cambridge University Press

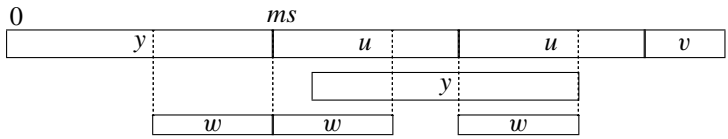
Solution

Let ms be the starting position and p be the period of the maximal suffix of x. The solution consists in checking the condition at line 2 below, which takes less than $|x|/2$ comparisons and answers the question.

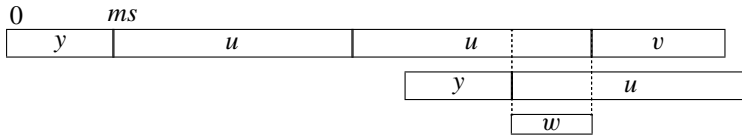
```
PERIODIC(x non empty word)
1  (ms, p) ← MAXSUFFIXPP(x)
2  if ms < |x|/2 and p ≤ |x|/2
       and x[0..ms − 1] suffix of x[ms..ms + p − 1] then
3    return TRUE
4  else return FALSE
```

It is clear that if the condition holds x is periodic with period p and if $ms \geq |x|/2$ or $p > |x|/2$ the word is not periodic.

Let $x = yz$, where $y = x[0..ms-1]$, $z = x[ms..|x|-1] = u^k v$ is the maximal suffix, u the prefix period of z, v a proper prefix of u and $k > 0$. Assume y is not a suffix of u.



We first consider the case where $|y| \geq |u|$ and show that $per(x) > |z|$. If not (above picture), a second occurrence of y in x overlaps some occurrence of u. Let w be their overlap, suffix of y and prefix of u (or v) and then of z. Let $z = wz'$. Both wz and z' are suffixes of x smaller than z. But $wz < z = wz'$ implies $z < z'$, a contradiction. In this situation, we have $per(x) > |z|$ and $per(x) > |y|$ (see Problem 41), which yields $2per(x) > |y| + |z|$ and proves that x is not periodic.



In the second case $|y| < |u|$ and we first show $per(x) > \min\{|z|, |x| - |v|\}$. If not and $per(x) \leq |z|$ we get the same conclusion as above. We then consider by contradiction that $per(x) \leq |x| - |v|$. In fact we have $per(x) < |x| - |v|$ because y is not suffix of u. Thus u (strictly) overlaps itself (picture below). A contradiction because u is border-free (see Problem 40 for example). In this

https://doi.org/10.1017/9781108835831.004 Published online by Cambridge University Press

situation, we have $per(x) > |x| - |v|$ and also trivially $per(x) > |v|$ then $2per(x) > |x|$, which shows that x is not periodic. This ends the proof of correctness of PERIODIC.

Notes

Algorithm PERIODIC tests the periodicity of x but does not compute its period. In fact it is possible to compute the period with the same time and space complexities using the time–space optimal string matching in [69]. The time–space optimal algorithm by Galil and Seiferas [124] (see [97]) can certainly be tuned to yield the same result as well.

25 Strict Borders

When used for searching texts online, the border table of a pattern is better replaced by the notion introduced in this problem. The effect is to improve the behaviour of searches as shown in Problem 26.

The ***strict-border table*** of a non-empty word x is defined on the lengths ℓ, $\ell = 0, \ldots, |x|$, of its prefixes by: $stbord[0] = -1$, $stbord[|x|] = border[|x|]$ and, for $0 < \ell < |x|$, $stbord[\ell]$ is the greatest t satisfying

- $-1 \leq t < \ell$ and
- ($x[0\,..\,t-1]$ is a border of $x[0\,..\,\ell-1]$ and $x[t] \neq x[\ell]$) or $t = -1$.

Word $x[0\,..\,stbord[\ell]-1]$ is the strict border of prefix $x[0\,..\,\ell-1]$ of x. It exists only if $stbord[\ell] \neq -1$.

Here are border and strict-border tables of the word `abaababaaba`:

i		0	1	2	3	4	5	6	7	8	9	10
$x[i]$		a	b	a	a	b	a	b	a	a	b	a
ℓ	0	1	2	3	4	5	6	7	8	9	10	11
$border[\ell]$	−1	0	0	1	1	2	3	2	3	4	5	6
$stbord[\ell]$	−1	0	−1	1	0	−1	3	−1	1	0	−1	6

https://doi.org/10.1017/9781108835831.004 Published online by Cambridge University Press

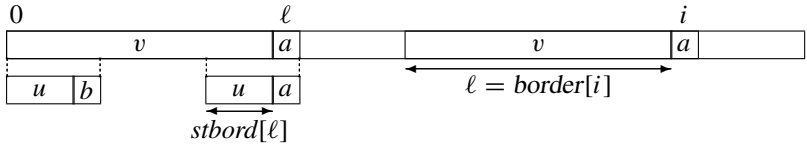
Question. Design a linear-time algorithm computing the table *stbord* without using the table *border* or any other additional table.

Solution

First notice that the table *stbord* can be used to compute the table *border*. It consists just in substituting *stbord* to *border* in the instruction at line 5 of Algorithm BORDERS (see Problem 19), which gives

```
BORDERS(x non-empty word)
1  border[0] ← −1
2  for i ← 0 to |x| − 1 do
3      ℓ ← border[i]
4      while ℓ ≥ 0 and x[i] ≠ x[ℓ] do
5          ℓ ← stbord[ℓ]
6      border[i + 1] ← ℓ + 1
7  return border
```

This accelerates the computation of the table on examples like $\mathtt{a}^n\mathtt{b}$.



Observation. The computation of the table of strict borders is based on the following property. If $\ell = border[i]$ for some i, $0 \le i \le |x|$, then

$$stbord[i] = \begin{cases} \ell & \text{if } i = 0 \text{ or } i = |x| \text{ or } x[i] \neq x[\ell], \\ stbord[\ell] & \text{else } (0 < i < |x| \text{ and } x[i] = x[\ell]). \end{cases}$$

Indeed, if the first condition is met the border of $x[0\,..\,i-1]$ complies with the definition of a strict border; then $stbord[i] = border[i]$. If not (see picture), the situation is the same as when computing the strict border of $x[0\,..\,\ell]$, then $stbord[i] = stbord[\ell]$.

https://doi.org/10.1017/9781108835831.004 Published online by Cambridge University Press

```
StrictBorders(x non-empty word)
 1  stbord[0] ← −1
 2  ℓ ← 0
 3  for i ← 1 to |x| − 1 do
 4       ▷ here ℓ = border[i]
 5       if x[i] = x[ℓ] then
 6            stbord[i] ← stbord[ℓ]
 7       else stbord[i] ← ℓ
 8            do  ℓ ← stbord[ℓ]
 9            while ℓ ≥ 0 and x[i] ≠ x[ℓ]
10       ℓ ← ℓ + 1
11  stbord[|x|] ← ℓ
12  return stbord
```

Algorithm StrictBorders solves the problem. The instructions at lines 5–7 implement the observation. Instructions at lines 8–9 correspond to those at lines 4–5 in Algorithm Borders and serve to compute the length ℓ of the border of $x[0\,..\,i+1]$. The correctness follows from these remarks and the linear running time follows from that of Algorithm Borders.

Notes

The table *stbord* is part of the design of the string-matching algorithm by Knuth, Morris and Pratt [162], which improves on the initial algorithm by Morris and Pratt (see [74, chapter 2]). For this online algorithm the improvement is on the delay between processing two consecutive symbols of the searched text (see Problem 26). A further improvement on the delay is provided by String matching automata (see Problem 27).

https://doi.org/10.1017/9781108835831.004 Published online by Cambridge University Press

26 Delay of Sequential String Matching

Algorithm KMP encapsulates a key feature (the use of a border table or the like) for the design of string-matching algorithms processing the text sequentially. Its concept is used for various types of patterns after appropriate preprocessing.

Algorithm KMP searches a text for occurrences of a pattern x. After preprocessing x it treats the text in an online manner and outputs found occurrences. It runs in linear time executing no more letter comparisons than twice the text length. In the version below, it just outputs '1' each time an occurrence of x is found in the text and outputs '0' otherwise.

The algorithm runs in linear time but not in real time due to the internal while loop of the algorithm dealing with a symbol of the text and that can take some time. This is called the *delay* of the algorithm. The goal of the problem is to bound the delay, precisely defined as the maximum number of comparisons executed at line 4 on the letter a of the input *text*.

KMP(x, *text* non-empty words)

```
1  stbord ← StrictBorders(x)
2  i ← 0
3  for each letter a of text, sequentially do
4      while (i = |x|) or (i ≥ 0 and a ≠ x[i]) do
5          i ← stbord[i]
6      i ← i + 1
7      if i = |x| then output 1 else output 0
```

If the table *border* (see Problem 19) of x is used in the algorithm instead of its table *stbord* (see Problem 25), the delay is $|x|$ in the worst-case. For example, if $x = \mathtt{a}^m$ is aligned with a factor $\mathtt{a}^{m-1}\mathtt{b}$ of the text, the letter $\mathtt{b}$ is compared to all the letters of x. But with the table *stbord* the delay becomes logarithmic.

Question. Show the delay of Algorithm KMP is $\Theta(\log |x|)$ when searching for a word x.

[**Hint:** Consider interlocked periods and apply the Periodicity Lemma.]

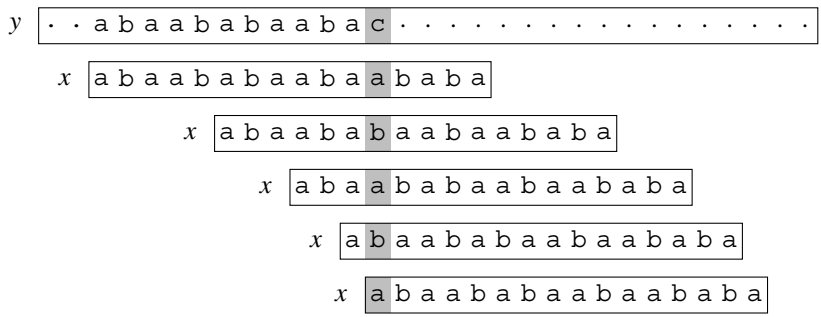
https://doi.org/10.1017/9781108835831.004 Published online by Cambridge University Press

Solution

Delay lower bound. The worst-case delay $\Omega(\log |x|)$ is reached, for example, when x is a prefix of a Fibonacci word.

Let x ($|x| > 2$) be such a word and let k be the integer for which $F_{k+2} \leq |x| + 1 < F_{k+3}$. The pattern x is a prefix of f_{k+1} (of length F_{k+3}) of the form $uabv$, where $uab = f_k$ for some letters $a, b \in \{\mathtt{a}, \mathtt{b}\}$. When ua is aligned with a factor uc of *text* and $c \notin \{\mathtt{a}, \mathtt{b}\}$, letter c is compared k times unsuccessfully alternatively to a and to b. Since k is of the order of $\log F_{k+2}$, then of the order of $\log |x|$, this gives the lower bound.

Example. Let x = abaababaabaababa, prefix of f_6 ($|f_6| = F_8$). We have $F_7 = 13$, $|x| + 1 = 17$, $F_8 = 21$. When the prefix abaababaabaa is aligned with the factor abaababaabac of the searched text, exactly $k = 5$ comparisons are done by Algorithm KMP before dealing with the letter following c (see picture).



Delay upper bound. For a position i on x let k be the largest integer for which both $stbord^{k-1}[i]$ is defined and $stbord^k[i]$ is not. We show that the integer k is an upper bound on the number of comparisons between $x[i]$ and a letter of the text.

Let us first show that if $stbord^2[i]$ is defined the prefix $u = x[0\,..\,i-1]$ satisfies $|u| \geq stbord[i] + stbord^2[i] + 2$. Since $stbord[i]$ and $stbord^2[i]$ are borders of $x[0\,..\,i-1]$, $p = |u| - stbord[i]$ and $q = |u| - stbord^2[i]$ are periods of u. By contradiction, if $|u| < stbord[i] + stbord^2[i] + 2$ then $p + q - 1 \leq |u|$. Thus, by the Periodicity Lemma, $q - p$ is also a period of u. This implies that $x[stbord^2[i]] = x[stbord[i]]$, letters at distance $q - p$ in u, which is a contradiction with the definition of *stbord*.

The inequality enables to show, by recurrence, that $|u| \geq F_{k+2} - 2$. Thus $|x| + 1 \geq |u| + 2 \geq F_{k+2}$. From the classical inequality $F_{n+2} \geq \Phi^n$ we get the delay upper bound of $O(\log_\Phi |x|)$.

Lower and upper bounds answer the question.

https://doi.org/10.1017/9781108835831.004 Published online by Cambridge University Press

Notes

The proof of the problem can be found in [162] (see also [74]). The algorithm by Simon [225] and by Hancart [136] reduces even more the upper bound on the delay to $\min\{\log_2 |x|, |alph(x)|\}$ using sparse matching automata (see Problem 27).

---

27 Sparse Matching Automaton

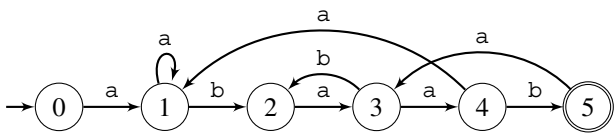
The most standard method to locate given patterns in a text processed sequentially is to use a pattern-matching automaton. Border tables used in Algorithm KMP may be viewed as implementations of such an automaton. The aim of the problem is to show another implementation technique that eventually improves searches. Searches using this implementation run at least as fast as Algorithm KMP, then run in linear time executing no more letter comparisons than twice the text length.

The pattern-matching or ***string-matching automaton*** $\mathcal{M}(x)$ of a word x drawn from the alphabet A is the minimal deterministic automaton accepting words ending with x. It accepts the language A^*x and has $|x| + 1$ states, $0, 1, \ldots, |x|$, the initial state 0 and its only accepting state $|x|$. Its transition table δ is defined, for a state i and a letter a, by

$$\delta(i,a) = \max\{s + 1 : -1 \leq s \leq i \text{ and } x[0\,..\,s] \text{ suffix of } x[0\,..\,i-1] \cdot a\}.$$

Observe that the size of the table is $\Omega(|x|^2)$ when the alphabet of x is as large as its length. But the table is very sparse, since most of its values are null.

Below is the string-matching automaton of `abaab` of length 5 on the alphabet $\{\mathtt{a}, \mathtt{b}\}$. Labelled arcs represent non-null values in the transition table. There are five forward arcs (on the main line) and four backward arcs. All other undrawn arcs have 0 as the target state. Were the alphabet to contain a third letter, all arcs labelled with it would also have 0 as the target state.



https://doi.org/10.1017/9781108835831.004 Published online by Cambridge University Press

Question. Show that the table δ associated with the string-matching automaton of a word of length n has at most $2n$ non-zero entries and that the bound is tight.

Solution

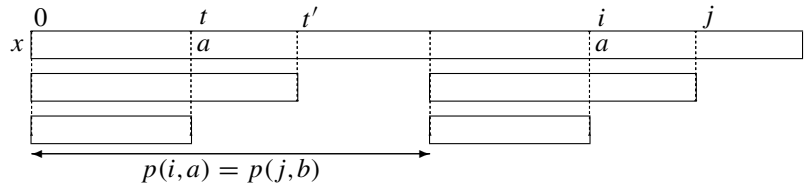
In the automaton $\mathcal{M}(x)$, there are n forward arcs corresponding to $\delta(i,a) = i+1$. The other arcs are backward arcs when $0 < \delta(i,a) \leq i$. Showing that there are at most n backward arcs answers the question.

Observation. An entry $\delta(i,a) = t$ associated with a backward arc satisfies $x[t] = a$, $x[i] \neq a$ and $x[0\,..\,t-1] = x[i-t\,..\,i-1]$. Then since the latter word is a border of length t of $x[0\,..\,i-1]$, $i-t$ is a period (not necessarily the smallest) of $x[0\,..\,i-1]$ denoted by $p(i,a)$.

For the above example, associated with the backward arcs we have $p(1,\mathtt{a}) = 1$, $p(3,\mathtt{b}) = 2$, $p(4,\mathtt{a}) = 4$ and $p(5,\mathtt{a}) = 3$. Note that $p(4,\mathtt{a}) = 4$ is not the smallest period of $x[0\,..\,3] = \mathtt{abaa}$.

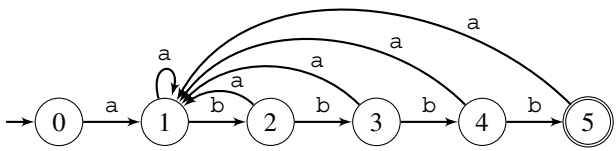
To prove the bound on the number of backward arcs we first show that p is a one-to-one function between backward arcs and periods of prefixes of x.

Let $\delta(i,a) = t+1$ with $0 \leq t < i$, $\delta(j,b) = t'$ with $0 \leq t' < j$. We assume $p(i,a) = p(j,b)$; that is, $i-t = j-t'$, and prove that $(i,a) = (j,b)$.



Indeed, if $i = j$ we get $t = t'$ and then $a = x[t] = x[t'] = b$; thus $(i,a) = (j,b)$. And otherwise, if for example $i < j$ like in the picture, since $i - t = p(i,a) = p(j,b)$ is a period of $x[0\,..\,j-1]$, we immediately get $x[t] = x[t+(i-t)] = x[i]$, which contradicts the definition of t.

Consequently p is a one-to-one function and since its values range from 1 to n, the number of backward arcs is no more than n, which achieves the first part of the proof.



https://doi.org/10.1017/9781108835831.004 Published online by Cambridge University Press

The bound in the statement is tight because the string-matching automaton of the word $\mathtt{ab}^{n-1}$ has exactly n backward arcs in addition to its n forward arcs.

Notes

The sparsity of the transition table of string-matching automata was observed by Simon [225]. The complete analysis is by Hancart [136] (see [74, chapter 2]), who showed how to use it for improving the string-matching algorithm by Knuth, Morris and Pratt [162]. The sparsity of the automaton does not extend to the similar automaton for a finite set of words.

The above result also applies to an analogue table used in the string-matching algorithm by Boyer and Moore (see [74, 98, 134]).

28 Comparison-Effective String Matching

When the access to textual data is through equality comparisons between symbols only, it is appropriate to reduce their number during searches for patterns. For example, Algorithm KMP (in Problem 26) executes in the worst case $2|y| - 1$ letter comparisons to find occurrences of a preprocessed pattern x in a text y. The same result holds when the search uses a string-matching automaton with an appropriate implementation.

The problem shows that the number of letter comparisons can easily be reduced to only $|y|$ comparisons for simple patterns. A similar approach, sketched at the end of the solution, leads to a maximum of $\frac{3}{2}|y|$ letter comparisons for general patterns.

Question. Design an algorithm searching a text y for all occurrences of a two-letter pattern x and using at most $|y|$ comparisons in the equality model.

[**Hint:** Distinguish whether the two letters are identical or not.]

Solution

We consider two cases, whether the pattern x is of the form aa or ab for two letters a and b with $a \neq b$.

https://doi.org/10.1017/9781108835831.004 Published online by Cambridge University Press

In the first case, Algorithm KMP executes exactly $|y|$ comparisons whichever border table is used. Note that a straightforward naive approach is likely to execute close to $2|y|$ comparisons.

In the second case, when $x = ab$ with $a \neq b$, the search is done with the following algorithm.

```
SEARCH-FOR-AB-IN(y word, a and b different letters)
1  j ← 1
2  while j < |y| do
3      if y[j] = b then
4          if y[j − 1] = a then
5              ab occurs at position j − 1 on y
6          j ← j + 2
7      else j ← j + 1
```

The algorithm can be viewed as a recursive algorithm:

find the smallest position $j > 0$ for which $y[j] = b$,
check if $y[j-1] = a$,
then recursively search for ab in $y[j+1\,..\,|y|-1]$.

Note that computing j during the first step of the recursive version takes exactly j comparisons since there is no comparison on $y[0]$.

We prove that Algorithm SEARCH-FOR-AB-IN makes at most $|y|$ symbol comparisons by induction on the length of y. Assume we found the first occurrence of b at position ℓ on y. This is done with ℓ comparisons. Then accounting for the comparison $y[\ell-1] = a$ gives a total of $\ell+1$ comparisons to deal with $y[0\,..\,\ell]$ of length $\ell+1$.

Since the same steps are applied to $y[\ell+1\,..\,|y|-1]$, by the inductive assumption the algorithm executes at most $|y| - \ell - 1 = |y[\ell+1\,..\,|y|-1]|$ more comparisons. Together we get at most $|y|$ comparisons as expected. This proves the inductive claim and completes the proof.

General patterns. To apply a similar technique for a general non-empty pattern x, this one is split as $x = a^k bu$, where u is a word, a and b are different letters and $k > 0$. The algorithm consists in searching for bu in the text (scanning it from left to right) and, whenever it makes sense, in checking if a^k occurs immediately before occurrences of bu. The search for bu can use, for example, the Morris-Pratt algorithm (Algorithm KMP with the table *border* instead of *stbord* in Problem 26). There are several versions implementing this method. However, it is quite technical and details are not included here.

https://doi.org/10.1017/9781108835831.004 Published online by Cambridge University Press

Notes

The first string-matching algorithm achieving a $\frac{3}{2}n$ ($n = |y|$) upper bound on letter comparisons is by Apostolico and Crochemore [14]. It has been improved to a $\frac{4}{3}n - \frac{1}{3}m$ comparisons bound by Galil and Giancarlo [123]. A tight upper bound of $n + \frac{8}{3(m+1)}(n-m)$ comparisons, where $m = |x|$, is proved by Cole and Hariharan in [60].

Under the additional constraint that searches operate strictly online on the text, the exact upper bound (obviously larger than above best bounds) on letter comparisons is $(2 - \frac{1}{m})n$ by Hancart [136] and by Breslauer et al. [44].

29 Strict Border Table of the Fibonacci Word

The border table *border* (see Problem 19) of the infinite Fibonacci word **f** has a simple structure but values in its strict border table *stbord* (see Problem 25) look chaotic at first glance. The problem examines a simple relation between the two tables, which helps to quickly compute any individual value of the table *stbord*.

Below are tables of periods, borders and strict borders related to a prefix of Fibonacci word. Values at index ℓ correspond to the prefix $\mathbf{f}[0\,..\,\ell-1]$ of length ℓ.

i		0	1	2	3	4	5	6	7	8	9	10
$x[i]$		a	b	a	a	b	a	b	a	a	b	a
ℓ	0	1	2	3	4	5	6	7	8	9	10	11
period[ℓ]		1	2	2	3	3	3	5	5	5	5	5
border[ℓ]	-1	**0**	0	**1**	1	2	**3**	2	3	4	5	**6**
stbord[ℓ]	-1	**0**	-1	**1**	0	-1	**3**	-1	1	0	-1	**6**

Question. Show how to compute in logarithmic time the nth element of the strict border table of the infinite Fibonacci word **f**.

[**Hint:** Examine positions where tables *border* and *stbord* match.]

https://doi.org/10.1017/9781108835831.004 Published online by Cambridge University Press

Solution

Positions ℓ on tables *border* and *stbord* pointing to the same entry are essential for the computation of $stbord[\ell]$. Other values in *stbord* can be computed from them.

The table *period* of periods of prefixes of $\mathbf{f}$ (defined for $\ell > 0$ by $period[\ell] = per(\mathbf{f}[0\,..\,\ell-1]) = \ell - border[\ell]$) has an extremely simple structure, encapsuled in the following observation.

Observation. Values in the table *period* of Fibonacci word can be seen as the word

$$1\,2\,2\,3\,3\,3\,5\,5\,5\,5\,5\,8\cdots = 1^1\,2^2\,3^3\,5^5\,8^8\,13^{13}\,21^{21}\cdots$$

composed of a concatenation of runs. Runs are all unary runs of a Fibonacci number with exponent equal to the same number and starting with $F_2 = 1$.

Notice that, for $\ell > 0$, the equality $stbord[\ell] = border[\ell]$ holds exactly at positions where the periodicity breaks, that is, at the end of a unary run. Let H be the increasing sequence of these (positive) positions. Hence, due to the observation on the structure of periods, the positions in H are

$$1,\ 1+2,\ 1+2+3,\ 1+2+3+5,\ 1+2+3+5+8,\ \ldots$$

and it is rather straightforward to see that

$$H = (1,3,6,11,19,32,\ldots) = (n > 0 : n+2 \text{ is a Fibonacci number}).$$

The relation between tables *border* and *stbord* (shown in Problem 25) can then be reformulated as

$$stbord[\ell] = \begin{cases} border[\ell] & \text{if } \ell \in H, \\ stbord[border[\ell]] & \text{otherwise.} \end{cases}$$

The formula translates immediately into Algorithm FibStrictBorders that computes $stbord[n]$ corresponding to the prefix of length n of the Fibonacci word. In the algorithm, the computation of $border[n] = n - period[n]$ is fast due to the structure of the sequence of periods. Testing if $n+2$ is a Fibonacci number can be done by keeping the greatest two Fibonacci numbers not larger than $n+2$. When going to smaller values in the recursion, the two Fibonacci numbers follow naturally.

https://doi.org/10.1017/9781108835831.004 Published online by Cambridge University Press

```
FibStrictBorders(n natural number)
1  if n = 0 then
2      return −1
3  elseif n + 2 is a Fibonacci number then
4      return n − period[n]
5  else return FibStrictBorders(n − period[n])
```

The following observation is the argument used to prove the running time of the algorithm.

Observation. $period[n]/n \geq \lim_{n\to\infty} \frac{F_{n-2}}{F_n} \geq \frac{1}{3}$.

As a consequence $n - period[n] \leq \frac{2}{3}n$, which implies that the depth of the recursion is logarithmic. Consequently the total running time of the algorithm is also logarithmic with respect to n.

30 Words with Singleton Variables

The problem shows the flexibility of the border table notion and of the fast algorithm computing this table. Having such a table is a valuable element to design efficient pattern matching, searching here for patterns with a variable.

We consider words over the alphabet $\mathtt{A} = \{\mathtt{a}, \mathtt{b}, \ldots\}$ in which letters are considered as singleton variables, that is, each letter represents a distinct unknown letter of the alphabet.

Two words u and v are said to be equivalent, denoted as $u \equiv v$, if there is a bijective letter-to-letter morphism $h : alph(u)^* \to alph(v)^*$ for which $h(u) = v$. For example $\mathtt{aacbaba} \equiv \mathtt{bbdabab}$ through the morphism h on $\mathtt{A}^*$ to itself defined by $h(\mathtt{a}) = \mathtt{b}$, $h(\mathtt{b}) = \mathtt{a}$, $h(\mathtt{c}) = \mathtt{d}$ and $h(\mathtt{d}) = \mathtt{c}$. When $alph(u) = alph(v)$ and $u \equiv v$ the words become equal after permuting their letters.

The pattern-matching problem is naturally redefined as follows: given a pattern word x and a text y, check if a factor z of y is equivalent to x: $z \equiv x$. For example, the pattern $x = \mathtt{aacbaba}$ occurs in $y = \mathtt{babbdababbacb}$ because its factor $z = \mathtt{bbdabab}$ is equivalent to x.

https://doi.org/10.1017/9781108835831.004 Published online by Cambridge University Press

Question. Assume the alphabet is sortable in linear time. Design a linear-time algorithm that searches a text for a pattern with singleton variables.

[**Hint:** Design a notion of border table adequate to the problem.]

Solution

The solution is based on the notion of a ***varying border table***. It is denoted by *vbord* and defined, for a parameter m and a non-empty word $w = w[0\,..\,n-1]$, on the lengths $\ell = 0, \ldots, n$ of its prefixes as follows: $vbord[0] = -1$ and, for $0 < j \leq n$, $vbord[j]$ is

$$\max\{t : 0 \leq t < \min\{j, m+1\} \text{ and } w[0\,..\,t-1] \equiv w[j-t+1\,..\,j]\}.$$

In other words, $vbord[j]$ is the length ℓ of a longest proper suffix of length at most m of $w[1\,..\,j]$ that is equivalent to its prefix of length ℓ. The reason for the restriction to length m appears when matching a pattern of length m.

Here is the table *vbord* for $w = \texttt{abaababbabba}$ and $m = 4$:

i		0	1	2	3	4	5	6	7	8	9	10	11
$w[i]$		a	b	a	a	b	a	b	b	a	b	b	a
ℓ	0	1	2	3	4	5	6	7	8	9	10	11	12
$vbord[\ell]$	−1	0	1	2	1	2	3	3	4	2	3	4	2

In the algorithm that builds *vbord* another table named *pred* is used and defined, for $0 \leq i < n$, by

$$pred[i] = \max\{t : t < i \text{ and } w[i] = w[t]\} \cup \{-1\}.$$

For example, if $w = \texttt{abcaabac}$ we get $pred = [-1, -1, -1, 0, 3, 1, 4, 2]$.

Observation 1. Table *pred* can be computed in linear time.

Let ∇_m denote the following predicate: $\nabla_m(i, \ell) = \text{TRUE}$ if and only if

$$(pred[\ell] \geq 0 \;\&\; i = pred[i] + k) \text{ or } (pred[\ell] = -1 \;\&\; pred[i] < i - \ell),$$

where $k = \ell - pred[\ell]$.

The proof of the next technical simple fact is left to the reader.

Observation 2. Assume $w[0\,..\,\ell-1] \equiv w[i-\ell\,..\,i-1]$ and $\ell < m$. Then $w[0\,..\,\ell] \equiv w[i-\ell\,..\,i] \iff \nabla_m(i, \ell)$.

In the classical algorithm computing border tables (see Problem 19) we can replace the test for symbols inequality by ∇_m. In this way we get the following linear-time algorithm computing the varying border table. Its correctness follows from Observation 2.

https://doi.org/10.1017/9781108835831.004 Published online by Cambridge University Press

```
VarBorders(x non-empty word)
 1  vbord[0] ← −1
 2  for i ← 0 to |x| − 1 do
 3       ℓ ← vbord[i]
 4       while ℓ ≥ 0 and not ∇_m(i, ℓ) do
 5            ℓ ← vbord[ℓ]
 6       if ℓ < m then
 7            vbord[i + 1] ← ℓ + 1
 8       else vbord[i + 1] ← vbord[ℓ + 1]
 9  return vbord
```

How it is related to pattern matching. The pattern-matching question, searching a text y for occurrences of the pattern x of length m, is solved with table *vbord*. This varying border table is built for the word $w = xy$ and parameter m. Then

$$x \equiv y[i - m + 1 \,..\, i] \iff vbord[m + i - 1] = m.$$

Hence searching y for pattern x with singleton variables reduces to the computation of the table *vbord*.

Notes

The problem here is a simplified version of the so-called parameterised pattern matching; see [24]. In this more general problem some symbols stands for singleton variables and some symbols are just constant letters. This is the subject of Problem 32.

https://doi.org/10.1017/9781108835831.004 Published online by Cambridge University Press

31 Order-Preserving Patterns

Searching time series or list of values for patterns representing specific fluctuations of the values requires a redefinition of the notion of pattern. The question is to deal with the recognition of peaks, breakdowns, double-dip recessions or more features on expenses, rates or the like.

In the problem we consider words drawn from a linear-sortable alphabet Σ of integers. Two words u and v of the same length over Σ are said to be *order-equivalent*, written $u \approx v$, if

$$u[i] < u[j] \iff v[i] < v[j]$$

for all pairs of positions i, j on the words. For example,

$$5\ 2\ 9\ 4\ 3 \approx 6\ 1\ 7\ 5\ 2,$$

which shows in particular their central value is the largest in both words.

The order-preserving pattern-matching problem is naturally defined as follows: given a pattern $x \in \Sigma^*$ and a text $y \in \Sigma^*$, check if x is order-equivalent to some factor of y. For example, word `5 2 9 4 3` appears equivalently at position 1 on

$$4\ \underline{6\ 1\ 7\ 5\ 2}\ 9\ 8\ 3$$

but nowhere else. For instance, it does not appear at position 4 because $5 < 8$, while in the pattern the corresponding values satisfy $5 > 4$.

For simplicity we assume that letters in each considered word are pairwise distinct (words are permutations of their set of letters).

Question. Design a linear-time algorithm for the order-preserving pattern-matching.

[**Hint:** Design a notion of border table adequate to the problem.]

Solution

The present solution is based on the notion of an ***OP-border table***. For the non-empty word $w = w[0\,..\,n-1] \in \Sigma^*$, the table *opbord* is defined by $opbord[0] = -1$ and, for $0 < \ell \leq n$, by $opbord[\ell] = t$, where $t < \ell$ is the largest integer for which $w[0\,..\,t-1] \approx w[\ell - t + 1\,..\,\ell]$.

Below is table *opbord* associated with $w =$ `1 3 2 7 11 8 12 9`.

https://doi.org/10.1017/9781108835831.004 Published online by Cambridge University Press

i		0	1	2	3	4	5	6	7
$w[i]$		1	3	2	7	11	8	12	9
ℓ	0	1	2	3	4	5	6	7	8
$opbord[\ell]$	-1	0	1	1	2	2	3	4	3

Two additional tables associated with w are defined to deal with the problem:

$$LMax[i] = j, \text{ where } w[j] = \max\{w[k] : k < i \text{ and } w[k] \leq w[i]\}$$

and $LMax[i] = -1$ if there no such $w[j]$,

$$LMin[i] = j, \text{ where } w[j] = \min\{w[k] : k < i \text{ and } w[k] \geq w[i]\}$$

and $LMin[i] = -1$ if there no such $w[j]$.

Observation 1. Both tables *LMax* and *LMin* can be computed in linear time.

Let us redefine the predicate ∇ (introduced in Problem 30) as follows:

$$\nabla_n(i, \ell) = 1 \iff w[p] \leq w[i] \leq w[q],$$

where $p = LMax[\ell]$ and $q = LMin[\ell]$ (if p or q equals -1 then the respective inequality is satisfied in the vacuum).

We leave to the reader the simple but technical proof of the following fact (see Notes).

Observation 2. Assume $w[0\,..\,\ell - 1] \approx w[i - \ell\,..\,i - 1]$ and $\ell < n$. Then

$$w[0\,..\,\ell] \approx w[i - \ell\,..\,i] \iff \nabla_n(i, \ell).$$

Pattern matching. To conclude, the algorithm to check if w is order-equivalent to some factor of a text is the same as for matching a pattern with singleton variables in Problem 30 except that the predicate ∇ is the predicate defined above.

Notes

The present algorithm is a version of the order-preserving pattern matching by Kubica et al. in [170], where Observation 2 is proved (see also [54, 139, 160]). The problem together with the possibility of mismatches is treated by Gawrychowski and Uznanski in [129]. Suffix trees for order-preserving indexing are introduced in [81].

https://doi.org/10.1017/9781108835831.004 Published online by Cambridge University Press

32 Parameterised Matching

The problem considers a more general and more complex version of Problem 30 where some symbols are unknown and some others are fixed constant symbols. Searching texts for a fixed pattern is rather restrictive in some contexts, and parameterised string matching provides an efficient solution in several applications by introducing variables in patterns. The problem was initially stated to detect code duplicates in which, for example, identifiers are substituted for the original names.

Let A and V be two disjoint alphabets: A is the alphabet of constant letters and V is the alphabet of variable letters. We assume that no alphabet contains integers. A word over $A \cup V$ is called a ***parameterised word*** or a p-word. Two p-words x and y are said to match or p-match if x can be transformed into y by applying a one-to-one mapping on symbols of V occurring in x.

For example, with $A = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ and $V = \{t, u, v, w\}$, $\mathtt{a}u\mathtt{b}v\mathtt{a}u\mathtt{b}$ and $\mathtt{a}w\mathtt{b}u\mathtt{a}w\mathtt{b}$ p-match by mapping u to w and v to u. But $\mathtt{a}u\mathtt{b}v\mathtt{a}u\mathtt{b}$ and $\mathtt{a}v\mathtt{b}w\mathtt{a}z\mathtt{b}$ do not p-match, since u should be mapped to both v and z.

The parameterised pattern matching problem can be stated as follows: given a pattern $x \in (A \cup V)^*$ and a text $y \in (A \cup V)^*$ find all the p-occurrences of x in y, that is, find all the positions j on y, $0 \le j \le |y| - |x|$, for which x and $y[j \,..\, j + |x| - 1]$ p-match.

For instance, with $y = \mathtt{a}z\mathtt{b}u\mathtt{a}z\mathtt{b}z\mathtt{a}v\mathtt{b}w\mathtt{a}v\mathtt{b}$ the pattern $x = \mathtt{a}u\mathtt{b}v\mathtt{a}u\mathtt{b}$ occurs at position 0 by mapping u to z and v to u and at position 8 by mapping u to v and v to w.

Question. Design an algorithm that solves the parameterised pattern matching problem and runs in linear time for a fixed alphabet.

Solution

The problem can be solved by adapting Algorithm KMP (see Problem 26) after a careful encoding of variables.

For a word $x \in (A \cup V)^*$ let $prev(x)$ be the word $z \in (A \cup \mathbf{N})^*$ defined, for a position i on x, by

$$z[i] = \begin{cases} x[i] & \text{if } x[i] \in A \\ 0 & \text{if } x[i] \in V \text{ not in } x[0 \,..\, i-1]. \\ i - \max\{j < i : x[j] = x[i]\} & \text{if } x[i] \in V \end{cases}$$

For instance, $prev(\mathtt{a}u\mathtt{b}v\mathtt{a}u\mathtt{b}) = \mathtt{a0b0a4b}$. The word $prev(x)$ can be computed in time $O(|x| \times \min\{\log |x|, \log |V|\})$ and $O(|x|)$ space, which reduces to $O(|x|)$ time and $O(|V|)$ space if V is a fixed alphabet.

https://doi.org/10.1017/9781108835831.004 Published online by Cambridge University Press

Let $z_i = z[i\,..\,|z|-1]$ be a suffix of $z \in (A \cup \mathbf{N})^*$. Then, $shorten(z_i)$ is the word s defined, for $0 \le j \le |z_i|-1$, by $s[j] = z_i[j]$ if $z_i[j] \le j$, and by $s[j] = 0$ otherwise. For instance, with $z = \mathtt{a0b0a4b}$, $z_3 = \mathtt{0a4b}$ and $shorten(z_3) = \mathtt{0a0b}$.

Observation. Let $z = prev(x)$. Letter $x[i] \in V$ p-matches the letter $y[j] \in V$ in y if one of the two conditions holds:

- $z[i] = 0$.
- $z[i] \neq 0$ and $y[j - z[i]] = y[j]$.

Due to the observation, p-matching x at a position j on y takes no long than $O(|x|)$ time. Mimicking Algorithm KMP with a ***parameterised-border table*** *pbord* gives a linear-time solution. For $x \in (A \cup V)^*$, it is defined by $pbord[0] = -1$ and, for $1 \le i \le |x|$, $pbord[i] = j$, where $j < i$ is the largest integer for which $prev(x[0\,..\,j-1])$ is a suffix of $shorten(prev(x[i-j\,..\,i-1]))$. Tables *prev* and *pbord* for $x = \mathtt{a}u\mathtt{b}v\mathtt{a}u\mathtt{b}$:

i	0	1	2	3	4	5	6	7
$x[i]$	a	u	b	v	a	u	b	
$prev(x)[i]$	a	0	b	0	a	4	b	
$pbord[i]$	−1	0	0	0	0	1	2	3

Given *pbord* for the p-word x, the next algorithm reports all positions of an occurrence of x in the word y.

```
ParameterisedMatching(x, y ∈ (A ∪ V)*)
 1  z ← prev(x)
 2  i ← 0
 3  for j ← 0 to |y| − 1 do
 4      while i ≥ 0 and not ((x[i], y[j] ∈ A and x[i] = y[j])
            or (x[i], y[j] ∈ V and
           (z[i] = 0 or y[j − z[i]] = y[j]))) do
 5          i ← pbord[i]
 6      i ← i + 1
 7      if i = |x| then
 8          report an occurrence of x at position j − |x| + 1
 9          i ← pbord[i]
```

https://doi.org/10.1017/9781108835831.004 Published online by Cambridge University Press

The proofs of correctness and of the complexity analysis of ParameterisedMatching are similar to the ones of Algorithm KMP. And the parameterised border table *pbord* can be computed by adapting Algorithm Borders that computes the usual border table (see Problem 19).

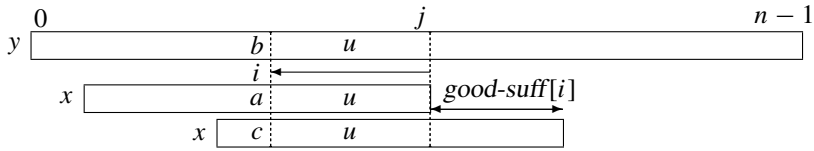
Notes

Parameterised pattern matching was first formalised by B. Baker [23, 24]. She introduced a solution based on Suffix trees for the offline version of the problem. A first solution for the online version was given in [11]. The present solution was first published in [145] together with a solution for online multiple parameterised pattern matching. The reader can refer to the survey that appeared in [188].

33 Good-Suffix Table

The Boyer–Moore algorithm (BM in Problem 34) applies the sliding window strategy on the text to locate occurrences of a pattern. It requires a pattern preprocessing to accelerate the search.

At a given step, the algorithm compares the pattern and a window on the text by computing their longest common suffix u. If $u = x$ a match occurs. Otherwise, in the generic situation, pattern $x[0\,..\,m-1]$ is aligned with the window, factor $y[j-m+1\,..\,j]$ of the text, au is a suffix of x and bu a suffix of the window, for different letters a and b.



To continue the search, Algorithm BM slides the window according to the period of x in case of a match or otherwise to the factor bu of the text to

https://doi.org/10.1017/9781108835831.004 Published online by Cambridge University Press

avoid positions of the window where no occurrence of x is possible. To do so it uses the ***good-suffix table*** *good-suff* defined for a position i on x and $u = x[i+1\,..\,m-1]$ by

$$good\text{-}suff[i] = \min\{|v| : x \text{ suffix of } uv \text{ or } cuv \text{ suffix of } x, c \neq x[i]\}.$$

The condition 'cuv suffix of x' with $c \neq a = x[i]$ ensures that when letter c is aligned with letter $b = y[j-m+1+i]$ after sliding the window, the same mismatch does not re-occur immediately (see picture). From the definition note that $good\text{-}suff[0] = per(x)$.

Question. Design a linear-time algorithm for computing the good suffix table of a word.

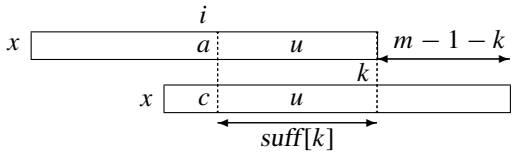
[**Hint:** Use the reverse table of prefixes of x^{R}.]

Solution

The solution uses the table of suffixes of x, *suff*, symmetric of the table of prefixes (see Problem 22), defined for a position i by $suff[i] = |lcs(x[0\,..\,i], x)|$, where *lcs* denotes the ***longest common suffix*** between x and $x[0\,..\,i]$.

Example. Tables *suff* and *good-suff* of `baacababa` are

i	0	1	2	3	4	5	6	7	8
$x[i]$	b	a	a	c	a	b	a	b	a
$suff[i]$	0	2	1	0	1	0	3	0	9
$good\text{-}suff[i]$	7	7	7	7	7	2	7	4	1



Observation. Tables *good-suff* and *suff* are closely related (see picture, in which $i = m - 1 - suff[k]$): $good\text{-}suff[m-1-suff[k]] \leq m-1-k$.

Then the computation of table *good-suff* is a mere application of the inequality and done by Algorithm GoodSuffixes. The get the smallest value of $m-1-k$, *suff* is scanned in increasing order of positions k (lines 8–9) after the table is filled in by periods of x (lines 3–7).

https://doi.org/10.1017/9781108835831.004 Published online by Cambridge University Press

```
GoodSuffixes(x non-empty word, suff its table of suffixes)
 1  m ← |x|
 2  p ← 0
 3  for k ← m − 2 to −1 do
 4      if k = −1 or suff[k] = k + 1 then
 5          while p < m − 1 − k do
 6              good-suff[p] ← m − 1 − k
 7              p ← p + 1
 8  for k ← 0 to m − 2 do
 9      good-suff[m − 1 − suff[k]] ← m − 1 − k
10  return good-suff
```

The overall computation takes $O(|x|)$ time, since the table *suff* can be computed in linear time (like table *pref* in Problem 22) and the above algorithm also runs in linear time when *suff* is given.

Notes

Table *good-suff* is often associated with a heuristics to account for the mismatch letter b as proposed by Boyer and Moore [41] (see also [162]). In fact this can be done for most string-matching methods.

The first exact algorithm for computing the good suffix table was designed by Rytter [212].

Table *good-suff* is the essential element of Algorithm BM. However, its above definition does not accurately use the mismatch letter b. This can be done using techniques related to sparse matching automata (see Problem 27) within space $O(|x|)$ independently of the alphabet size.

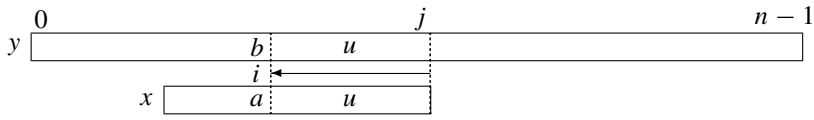
Table *suff* is used in a more efficient variant of Algorithm BM by Apostolico and Giancarlo [16] for which the maximal number of letter comparisons is $1.5|y|$ (see [89]).

https://doi.org/10.1017/9781108835831.004 Published online by Cambridge University Press

34 Worst Case of the Boyer–Moore Algorithm

Boyer–Moore string matching is based on a technique that leads to the fastest searching algorithms for fixed patterns. Its main feature is to scan the pattern backward when aligned with a factor of the searched text. A typical pattern preprocessing is shown in Problem 33.

When locating a fixed pattern x of length m in a text y of length n, in the generic situation x is aligned with a factor (the window) of y ending at position j (see picture). The algorithm computes the longest common suffix (*lcs*) between x and the factor of y, and, after possibly reporting an occurrence, slides the window towards the end of y based on the preprocessing and on information collected during the scan, without missing an occurrence of x. Algorithm BM implements the method with table *good-suff* of Problem 33:



BM(x, y non-empty words, m, n their lengths)

```
1  j ← m − 1
2  while j < n do
3      i ← m − 1 − |lcs(x, y[j − m + 1 .. j])|
4      if i < 0 then
5          report an occurrence of x at position j − m + 1 on y
6          j ← j + per(x)   ▷ per(x) = good-suff[0]
7      else j ← j + good-suff[i]
```

After position j on y is treated, if an occurrence of x is found the algorithm slides naturally the window at distance $per(x)$. If no occurrence is found, the distance $good\text{-}suff[i]$ depends on the factor bu of y (it depends on au of x in Problem 33). Value $per(x)$ and array *good-suff* are preprocessed before the search.

Question. Give examples of a non-periodic pattern and of a text y for which Algorithm BM performs close to $3|y|$ letter comparisons at line 3 for computing the longest common suffix.

Solution

Let $x = \mathtt{a}^{k-1}\mathtt{b}\mathtt{a}^{k-1}$ and $y = \mathtt{a}^{k-1}(\mathtt{aba}^{k-1})^{\ell}$ with $k \geq 2$. Then $m = 2k - 1$ and $n = \ell(k + 1) + (k - 1)$.

https://doi.org/10.1017/9781108835831.004 Published online by Cambridge University Press

Example. Let $k = 5$ and $\ell = 4$ when considering the pattern $\mathtt{a}^4\mathtt{b}\mathtt{a}^4$ of length 9 and the text $\mathtt{a}^4(\mathtt{aba}^4)^4$ of length 28. The picture illustrates the beginning of the search, which overall executes $4 \times 13 = 52$ letter comparisons.

Consider the position $j = (k-3) + p(k+1)$ on y with $p \geq 1$. We have $y[j-m+1\,..\,j] = \mathtt{a}^k\mathtt{b}\mathtt{a}^{k-2}$ and $|lcs(x, y[j-m+1\,..\,j])| = k-2$ computed with $k-1$ letter comparisons. The window slide length is $good\text{-}suff[m-k-1] = 1$, updating j to $(k-2) + p(k+1)$, and $y[j-m+1\,..\,j]$ becomes $\mathtt{a}^{k-1}\mathtt{b}\mathtt{a}^{k-1}$. This time $|lcs(x, y[j-m+1\,..\,j])| = m$ computed with m letter comparisons and the next slide length is $per(x) = k$, producing $j = (k-3) + (p+1)(k+1)$.

The two steps require $k-1+m = 3k-2$ comparisons and lead to a similar situation from which the same process repeats on each of the $\ell - 1$ first occurrences of the factor $\mathtt{aba}^{k-1}$ (of length $k+1$) of y. On the last occurrence $(k-1) + (k+1) = 2k$ comparisons are performed and on the prefix of length $k-1$ of y, $k-2$ comparisons are performed. Overall, Algorithm BM with these inputs x and y executes $\frac{3k-2}{k+1}(n-k+1) = \left(n - \frac{m-1}{2}\right)\left(3 - \frac{10}{m+3}\right)$ comparisons, as expected.

Notes

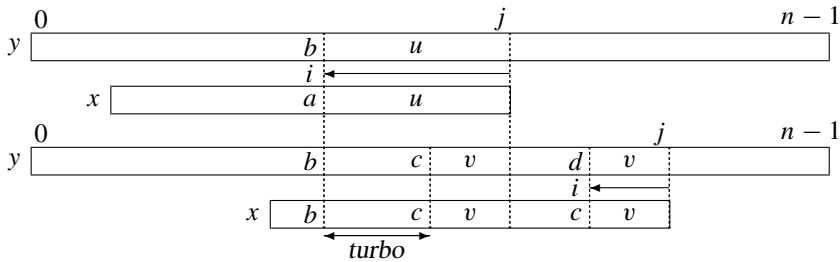
Boyer–Moore string matching is from [41] (see also [162]). The proof of the $3n$ comparison bound for searching for an aperiodic pattern in a text of length n is by Cole [59]. Detailed descriptions and variants of the Boyer–Moore algorithm can be found in classical textbooks on string algorithms [74, 96, 98, 134, 228].

https://doi.org/10.1017/9781108835831.004 Published online by Cambridge University Press

35 Turbo-BM Algorithm

The problem shows how a very light modification of Boyer–Moore string matching produces a much faster algorithm when the method is used to locate all the pattern occurrences in a text, like Algorithm BM in Problem 34 does. The initial design of the Boyer–Moore algorithm was to find the first pattern occurrence. It is known that for that goal it runs in linear time according to the searched text length. But it has a quadratic worst-case running time to report all pattern occurrences, especially for periodic patterns. This is due to its amnesic aspect when it slides the window to the next position.

The goal of Algorithm TURBO-BM is to get a linear-time search by adapting the Boyer–Moore search algorithm, Algorithm BM, without changing its pattern preprocessing, table *good-suff*. Only constant extra space is added and used during the search.



Algorithm TURBO-BM uses the lengths $mem = |u|$ of the previous suffix match u and $\ell = |v|$ of the current suffix match v to compute $\max\{good\text{-}suff[i], |u| - |v|\}$.

In the example below, the first match $u =$ `bababa` of length 6 leads to a slide of length $4 = good\text{-}suff[4]$. After sliding the window, the new match $v =$ `a` alone would give a slide of length $1 = good\text{-}suff[9]$. But the ***turbo-shift*** applies and produces a slide of length $turbo = 6 - 1 = 5$.

```
y   b a b a b b a b a b a a b a a . . . . .
x   b b a b a b a b a b a
                u

y   b a b a b b a b a b a a b a a . . . . .
x           b b a b a b a b a b a
                u       v
```

Question. Show that Algorithm TURBO-BM correctly reports all occurrences of the pattern x in y.

https://doi.org/10.1017/9781108835831.004 Published online by Cambridge University Press

```
TURBO-BM(x, y non-empty words, m, n their lengths)
 1  (j, mem) ← (m − 1, 0)
 2  while j < n do
 3      ▷ jumping over memory if mem > 0
 4      i ← m − 1 − |lcs(x, y[j − m + 1 .. j])|
 5      if i < 0 then
 6          report an occurrence of x at position j − m + 1 on y
 7          (shift, ℓ) ← (per(x), m − per(x))
 8      else (shift, ℓ) ← (good-suff[i], m − i − 1)
 9      turbo ← mem − (m − i − 1)
10      if turbo > shift then
11          (j, mem) ← (j + turbo, 0)
12      else (j, mem) ← (j + shift, ℓ)
```

Solution

Based on the proof of correctness of the original algorithm, that of Algorithm TURBO-BM depends on the fact that no occurrence of the pattern x in y is missed when the window is slid by *turbo* positions (line 11).

Indeed, when line 11 executes, $turbo > 1$ because $turbo > shift \geq 1$. Then cv is a (proper) suffix of u in x and in y, since the occurrences of u are aligned (see picture). The suffix $uzcv$ of x, z a word and c a letter, has period $|zcv|$ because u is a suffix of x. It is aligned with the factor $uz'dv$ of y, where z' is a word and d a letter. But since $c \neq d$, $|zcv|$ is not a period of the latter factor.

Therefore, the suffix $uzcv$ of x cannot cover both letters c and d occurring in y, which shows the ending position of the next possible occurrence of x in y is at least at position $j + turbo$ as required.

Notes

Several solutions have been proposed to cope with the quadratic-time behaviour of the Boyer–Moore algorithm. The first solution is by Galil [122] who showed a linear-time variant. Another solution by Apostolico and Giancarlo [16] requires extra memory space, linear in the pattern length, during both the preprocessing and the search steps and executes no more than $1.5n$ letter comparisons [89].

Algorithm TURBO-BM [70] is certainly the lightest improvement of the original algorithm. Not only does it provide a linear-time solution but also executes no more than $2n$ letter comparisons during the search step (see [74, 96]) at the cost of a constant extra memory space.

https://doi.org/10.1017/9781108835831.004 Published online by Cambridge University Press

36 String Matching with Don't Cares

Words in the problem are drawn from the alphabet of positive integers with an extra letter $*$. Letter $*$, called a don't care (or joker), stands for any other letter of the alphabet and matches any letter including itself.

Matching strings with don't cares consists in searching a text y for all occurrences of a pattern x, assuming the two words contain don't care symbols. Let $m = |x|$ and $n = |y|$.

Example. `ab*b` occurs in `abaaba*cbcb` at positions 3 and 5 only.

Contrary to several other string-matching algorithms, solutions to the present problem often use arithmetic operations for convolution: given sequences B and C of length at most n compute the sequence A of length $2n$ defined, for $0 \leq i \leq 2n-1$, by

$$A[i] = \sum_{j=0}^{n-1} B[j] \cdot C[i+j].$$

It is assumed that each elementary arithmetic operation executes in constant time and that convolution of sequences of length n can be done in $O(n \log n)$ time.

Question. Show how string matching with don't cares can be reduced in linear time to the convolution problem.

Solution

After changing the don't care symbol to zero, we define the sequence $A[0\,..\,n-m]$ by: $A[i] = \sum_{j=0}^{m-1} x|j] \cdot y[i+j] \cdot (x[j] - y[i+j])^2$, which is

$$\sum_{j=0}^{m-1} x[j]^3 y[i+j] - 2\sum_{j=0}^{m-1} x[j]^2 y[i+j]^2 + \sum_{j=0}^{m-1} x[j] \cdot y[i+j]^3.$$

The computation can then be done with three instances of convolution, running overall in $O(n \log n)$ time. The relation between A and the question stands in the next observation.

Observation. $A[i] = 0$ if and only if pattern x occurs at position i on y.

Indeed $A[i]$ is null if and only if each term $x|j] \cdot y[i+j] \cdot (x[j] - y[i+j])^2$ is null, which means that either $x[j]$ or $y[i+1]$ (originally equal to the don't care symbol) are null or both are equal. This corresponds to a match of letters and proves the correctness of the reduction algorithm.

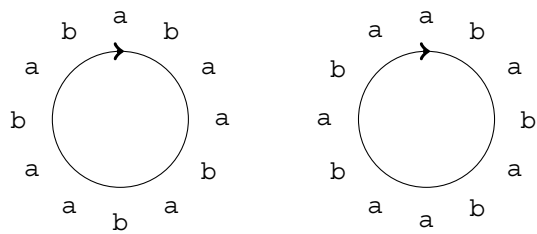
https://doi.org/10.1017/9781108835831.004 Published online by Cambridge University Press

Notes

Convolution adapted to string matching has been introduced by Masek and Paterson in [186]. The present simplification is from [58].

37 Cyclic Equivalence

Two words are cyclically equivalent if one is a conjugate (rotation) of the other. Testing their equivalence appears in some string matching questions but also in graph algorithms, for example, for checking the isomorphism of directed labelled graphs, in which the test applies to graph cycles.



The picture shows equivalent words drawn cyclically. The following algorithm tests cyclic equivalence using the alphabet ordering.

CyclicEquivalence(u, v non-empty words of length n)

```
 1  (x, i, y, j) ← (uu, 0, vv, 0)
 2  while i < n and j < n do
 3      k ← 0
 4      while k < n and x[i + k] = y[j + k] do
 5          k ← k + 1
 6      if k = n then
 7          return TRUE
 8      if x[i + k] > y[j + k] then
 9          i ← i + k + 1
10      else j ← j + k + 1
11  return FALSE
```

https://doi.org/10.1017/9781108835831.004 Published online by Cambridge University Press

Question. Show that Algorithm CyclicEquivalence checks if two words are cyclically equivalent and runs in linear time with constant extra space.

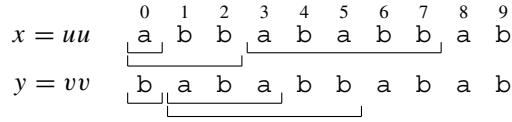
[**Hint:** Consider Lyndon conjugates.]

When the alphabet has no clear ordering, it is useful to have a solution based on simpler types of symbol comparisons.

Question. How would you test the cyclic equivalence of two strings with no ordering, that is, using only $=/\neq$ letter comparisons?

Solution

Let us run Algorithm CyclicEquivalence on words $u = \texttt{abbab}$ and $v = \texttt{babab}$ and look how pairs of indices (i, j) on $x = uu$ and $y = vv$ respectively evolve.



Starting from the pair of indices $(i, j) = (0, 0)$, after the first execution of instructions in the main while loop the pair becomes $(0, 1)$ because $x[0] < y[0]$. The algorithm then compares factor $x[0\,..\,2] = \texttt{abb}$ of x with factor $y[1\,..\,3] = \texttt{aba}$ of y, producing the next pair $(3, 1)$ because $x[2] > y[3]$. Eventually, in the next pass of the loop it detects that u and v are conjugate.

Commenting on the algorithm's run, note that indices i and j bump on the starting positions on x and y of the Lyndon conjugate ababb of the cyclically equivalent words u and v. (If u or v are not primitive, the argument applies to their roots.) The dramatic increase of i at line 9 or of j at line 10 intuitively comes from that property of Lyndon words: if wa is a prefix of a Lyndon word and letter a is smaller than letter b then wb is a Lyndon word. A consequence is that wb is border free. Therefore matching wa can resume right after an occurrence of wb. This is illustrated in particular when comparing aba and abb in the example.

Analysis of CyclicEquivalence. Let $u^{(k)}$ be the kth conjugate (kth rotation or kth-shift) of u, $k = 0, 1, \ldots, n-1$. For $x = uu$, $u^{(k)} = x[k\,..\,k+n-1]$. Similarly for $y = vv$, $v^{(k)} = y[k\,..\,k+n-1]$.

Let $D(u)$ and $D(v)$ be the set of positions on x and on y respectively:

$$D(u) = \{k : 0 \leq k < n \text{ and } u^{(k)} > v^{(j)} \text{ for some } j\},$$

$$D(v) = \{k : 0 \leq k < n \text{ and } v^{(k)} > u^{(i)} \text{ for some } i\}.$$

https://doi.org/10.1017/9781108835831.004 Published online by Cambridge University Press

The algorithm's correctness relies on the invariant of the main while loop: $[0\,..\,i-1] \subseteq D(u)$ and $[0\,..\,j-1] \subseteq D(v)$, which is easy to check because, for example, if we have $x[i+k] > y[j+k]$ at line 8, we have $x[i\,..\,i+k] > y[j\,..\,j+k]$, $x[i+1\,..\,i+k] > y[j+1\,..\,j+k]$, etc.

If the algorithm returns TRUE then $u^{(i)} = v^{(j)}$ and the two words are conjugate. If it returns FALSE, we have either $i = n$ or $j = n$. W.l.o.g., assuming $i = n$ we get $D(u) = [1\,..\,n]$. This means that every cyclic conjugate of u has a smaller conjugate of v. So the words cannot have the same smallest conjugate, which implies they are not conjugate.

The number of symbol comparisons is clearly linear. The largest number of comparisons is for conjugate words of the form $u = \mathtt{b}^k\mathtt{ab}^\ell$ and $v = \mathtt{b}^\ell\mathtt{ab}^k$. This implies linear running time. Modular arithmetic on indices can be used to avoid duplicating u and v, then reducing the extra space to a constant amount.

No ordering. A solution to solve the question without considering any ordering on the alphabet is to use a string-matching technique with this feature or the border table. For example, the border table of $u\#vv$, where # is a letter that does not occur in uv, allows locating u in vv. An occurrence exists if and only if u and v of the same length are conjugate.

Using a time–space optimal string-matching algorithms gives an overall algorithm sharing the features of CyclicEquivalence but that is far less simple and elegant than it.

Notes

The design of Algorithm CyclicEquivalence borrows ideas from the circular lists equivalence algorithm by Shiloach [223].

A less direct approach to solve cyclic equivalence is to use the function MaxSuffixPos (see Problems 38 and 40). After computing indices $i =$ MaxSuffixPos(uu) and $j =$ MaxSuffixPos(vv) that identify the maximal suffixes MaxSuffix(uu) and MaxSuffix(vv), the solution consists in testing the equality of their prefixes of length $|u| = |v|$.

The table *Lyn* (see Problem 87) can also be used but yields a less efficient technique.

Time–space optimal string-matching algorithms may be found in [94, 97, 124].

https://doi.org/10.1017/9781108835831.004 Published online by Cambridge University Press

38 Simple Maximal Suffix Computation

The maximal suffix of a word is its alphabetically greatest suffix. The notion is a key element in some combinatorial aspects of words (e.g. related to runs or critical positions) but also in the development of string-matching algorithms (e.g. the two-way algorithm used in some C libraries such as glibc and FreeBSD's lib). The algorithm presented in this problem is tricky but simpler than the one in Problem 40. Both work in place, that is, need only constant space to work in addition to their input (contrary to the solution in Problem 23), which makes their implementation straightforward.

For a non-empty word x, Algorithm MaxSuffixPos computes the starting position of MaxSuffix(x), ***maximal suffix*** of x. For example, MaxSuffixPos(bbabbbba) $= 3$ position of suffix bbbba of the input.

Note the similarity between MaxSuffixPos and Algorithm Cyclic-Equivalence in Problem 37 and the similarity of its pseudo-code with that of the other version in Problem 40.

MaxSuffixPos(x non-empty word)

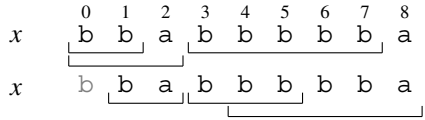
```
1  (i, j) ← (0, 1)
2  while j < |x| do
3      ▷ note the invariant i < j
4      k ← 0
5      while j + k < |x| and x[i + k] = x[j + k] do
6          k ← k + 1
7      if j + k = |x| or x[i + k] > x[j + k] then
8          j ← j + k + 1
9      else i ← i + k + 1
10     if i ≥ j then
11         j ← i + 1
12 return i
```

Question. Show that Algorithm MaxSuffixPos computes the position on a word of its maximal suffix and that it runs in linear time with constant extra space.

https://doi.org/10.1017/9781108835831.004 Published online by Cambridge University Press

Solution

Let us run Algorithm MaxSuffixPos on word $x = \texttt{bbabbbba}$, which is duplicated on the picture below to better show values of indices, i on the top row and j on the bottom row.



The first pair (i, j) of indices on x is $(0, 1)$, leading $x[0\,..\,1] = \texttt{bb}$ to be compared to $x[1\,..\,2] = \texttt{ba}$. This produces the next pair $(0, 3)$. Then the comparison between $x[0\,..\,2] = \texttt{bba}$ and $x[3\,..\,5] = \texttt{bbb}$ leads to increase i to 3 and additionally j to 4 to avoid equality. Eventually, j moves to 9, which causes the main loop and the procedure to stop. The algorithm returns position 3 of the maximal suffix bbbbba of bbabbbbba.

Correctness of MaxSuffixPos. The proof is just sketched, since its elements may be found in Problem 40. The proof relies on the following invariant of the main iteration (while loop):

> in the interval $[0\,..\,j-1]$ the position i is the only candidate as starting position of the maximal suffix of $x[0\,..\,j-1]$. In other words if $t \neq i$ and $t < j$ then t is not a starting position of the maximal suffix.

Hence, at the end j is not less than $|x|$ and i is the only possible position in the interval $[0\,..\,|x|-1]$ (all positions of x) starting the maximal suffix. Consequently i is the required output.

Notes

The present Algorithm MaxSuffixPos is a version of the algorithm by Adamczyk and Rytter [1].

The algorithm, after a simple cosmetic modification, also computes the shortest period of the maximal suffix as its other pseudo-code in Problem 40 does. Indeed, it is $i - j'$, where j' is the penultimate value of j (the last value of j is $|x|$, out of the range).

https://doi.org/10.1017/9781108835831.004 Published online by Cambridge University Press

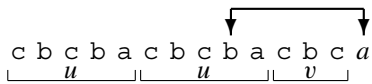
39 Self-Maximal Words

The notion of a ***self-maximal word***, word that is the alphabetically greatest among its suffixes, is somehow dual to the notion of Lyndon word, word that is smaller than all its proper non-empty suffixes. The former words appear naturally when locating critical positions on a word (see Problem 41) and in string-matching algorithms based on them.

Algorithm SELFMAXIMAL checks if its input word x is self-maximal, that is, if $x = \text{MaxSuffix}(x)$. It processes the word in real time since the instruction in the for loop executes in constant time.

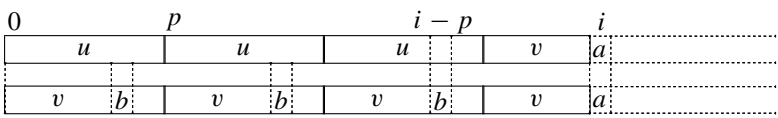
```
SELFMAXIMAL(x non-empty word)
1  p ← 1
2  for i ← 1 to |x| − 1 do
3      if x[i] > x[i − p] then
4          return FALSE
5      elseif x[i] < x[i − p] then
6          p ← i + 1
7  return TRUE
```

Example. The word $\texttt{cbcbacbcbacbc} = (\texttt{cbcba})^2\texttt{cbc}$ is self-maximal as well as its prefix period cbcba that is additionally border free. Its suffix cbcbacbc is essentially the only suffix competing with it as the greatest suffix. A letter a appended to it is then to be compared to the letter following the prefix occurrence of cbcbacbc.



Question. Prove that the above really simple algorithm correctly tests if its input word is greater than all its proper suffixes.

The next picture illustrates the role of variables in SELFMAXIMAL.



https://doi.org/10.1017/9781108835831.004 Published online by Cambridge University Press

Question. Upgrade the above algorithm to SMPrefix that computes the longest self-maximal prefix of a word.

Solution

A note before starting the proofs. Let y be a non-empty self-maximal word that is also border free, that is, $per(y) = |y|$. Then any proper non-empty suffix z of y satisfies $z \ll y$ (i.e., $z = ras$ and $y = rbt$ with letter a smaller than letter b). Therefore $zs' \ll yt'$ for any words s' and t'.

Correctness of SelfMaximal. Consider the invariant of the for loop: $u = x[0\,..\,p-1]$ is a non-empty self-maximal and border-free word, $x[0\,..\,i-1] = u^e v$, where $e > 0$ and v a proper prefix of u.

The invariant holds at the start of the loop because $u = x[0]$, $e = 1$ and v is the empty word. If the invariant holds at line 7, $x = u^e v$ and then x is self-maximal, as expected, with period $p = |u|$.

It remains to show that instructions within the loop do not change the invariant validity. We have three cases to consider according to the result of the letter comparison.

If $x[i] > x[i-p]$, x is not self-maximal, since its factor $x[p\,..\,i-1]x[i] = x[0\,..\,i-p-1]x[i]$ is greater than its prefix $x[0\,..\,i-p-1]x[i-p]$. If $x[i] = x[i-p]$, the invariant still holds, possibly with $e+1$ and $v = \varepsilon$.

The case in which $x[i] < x[i-p]$ provides the key feature of the algorithm: $x[0\,..\,i]$ becomes border free in addition to being self-maximal. To show it, let $a = x[i]$ and $b = x[i-p]$ with $a < b$. First consider suffixes of the form $v'a$ of va. Since vb is a prefix of the self-maximal u, $v'b \le u < x[p\,..\,i]$, and since $v'a < v'b$ we get $v'a < x[p\,..\,i]$. Second, suffixes $x'a$ starting at positions $j < e|u|$ that are not multiple of p are prefixed by a proper suffix u' of u. Referring to the above note, we have $u' \ll u$, which implies $x'a \ll u^e va = x[p\,..\,i]$. Third, the remaining suffixes $x'a$ to consider start at positions of the form $p, 2p, \ldots, ep$. All x' are also prefixes of u^e and followed by the letter b. Therefore $x'a < x'b < x[p\,..\,i]$. This completes the proof.

Algorithm SMPrefix. Its design follows closely that of SelfMaximal and its correctness readily comes from the above proof.

https://doi.org/10.1017/9781108835831.004 Published online by Cambridge University Press

```
SMPrefix(x non-empty word)
 1  p ← 1
 2  for i ← 1 to |x| − 1 do
 3      if x[i] > x[i − p] then
 4          return x[0 .. p − 1]
 5      elseif x[i] < x[i − p] then
 6          p ← i + 1
 7  return x[0 .. |x| − 1]
```

Notes

Algorithm SelfMaximal is adapted from the Lyndon factorisation algorithm by Duval [105] and reduced to exhibit its key feature.

40 Maximal Suffix and Its Period

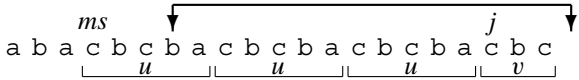
The maximal suffix of a word is its alphabetically greatest suffix. The problem follows Problem 38 and presents another pseudo-code for computing a maximal suffix. As the other it processes its input in linear time using only constant extra space.

```
MaxSuffixPP(x non-empty word)
 1  (ms, j, p, k) ← (0, 1, 1, 0)
 2  while j + k < |x| do
 3      if x[j + k] > x[ms + k] then
 4          (ms, j, p, k) ← (j, j + 1, 1, 0)
 5      elseif x[j + k] < x[ms + k] then
 6          (j, p, k) ← (j + k + 1, j − ms, 0)
 7      elseif k = p − 1 then
 8          (j, k) ← (j + k + 1, 0)
 9      else k ← k + 1
10  return (ms, p)
```

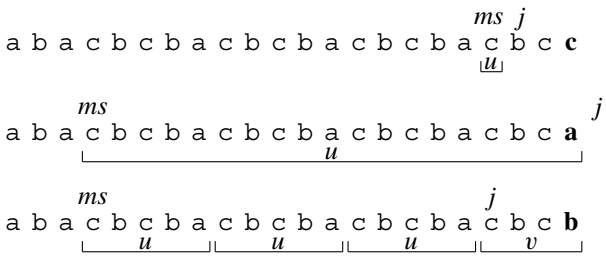
https://doi.org/10.1017/9781108835831.004 Published online by Cambridge University Press

Question. Show that Algorithm MaxSuffixPP computes the starting position and the period of the maximal suffix of its input word.

Example. $(\texttt{cbcba})^3\texttt{cbc} = \text{MaxSuffix}(\texttt{aba}(\texttt{cbcba})^3\texttt{cbc})$. A next letter is to be compared to the letter following prefix `cbc` of the maximal suffix.



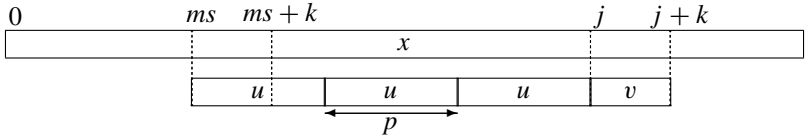
The pictures display the three possibilities corresponding to the execution of instructions at lines 4, 6 and 8 respectively.



Question. Show how to test the primitivity of a word in linear time and constant extra space.

Solution

Correctness of Algorithm MaxSuffixPP. Up to a change of variables the structure of MaxSuffixPP and its proof follow that of the algorithms in Problem 39. Here, *ms* stands for the starting position of MaxSuffix$(x[0\,..\,j+k-1]) = u^e v$ with u self-maximal and border free, $e > 0$ and v a proper prefix of u. The picture illustrates the role of variables i, j and k. The letter $x[j+k]$ is compared to $x[ms+k] = x[j+k-p]$.



Lines 3–4 correspond to the case when the candidate starting position *ms* of the maximal suffix has to be updated. Position j becomes the next candidate because suffix $x[j\,..\,j+k]$ is larger than suffixes of $x[ms\,..\,j+k]$ starting before j. And the process restarts from that position forgetting what has been done beyond j.

https://doi.org/10.1017/9781108835831.004 Published online by Cambridge University Press

Lines 5–6 deal with the key feature. The word $x[ms\,..\,j+k]$ is self-maximal and border free of period $j + k - ms + 1$, its length.

Eventually, lines 7–9 manage the incrementation of k, possibly updating j to keep k smaller than p.

Complexity of Algorithm MaxSuffixPP. Its memory space requirement is clear. Its running time is less obvious to evaluate due to its oblivious element at line 4. But it suffices to consider the value of the expression $ms + j + k$.

At line 4, ms is incremented by at least p, j by 1 and k decremented by at most $p - 1$, while p is unchanged; thus the value is strictly incremented. At line 6, j is incremented by $k + 1$ before k becomes null, leading again to a strict incrementation of the value. At lines 8–9, $j + k$ is incremented by 1 and ms unchanged, yielding the same conclusion.

Since $ms+j+k$ runs from 1 to at most $2|x|+1$, this shows that the algorithm stops and performs no more than $2|x|$ letter comparisons.

Primitivity test. Let $ms = \text{MaxSuffixPP}(x)$ and p be the period of MaxSuffix(x). It is known that $ms < p$ (see Problem 41). Let k be the largest integer for which $j = ms + kp \leq |x|$. Then testing primitivity amounts to check if $x[j\,..\,n-1]x[0\,..\,ms-1] = x[ms\,..\,ms+p-1]$.

Notes

Algorithm MaxSuffixPP is by Crochemore and Perrin in [94], where it is used for the preprocessing of a time–space optimal string matching, known as the two-way algorithm.

Relaxing the constant extra space aspect of the algorithm with a border table improves the running time of the technique but keeps the asymptotic behaviour unchanged in the worst case.

https://doi.org/10.1017/9781108835831.004 Published online by Cambridge University Press

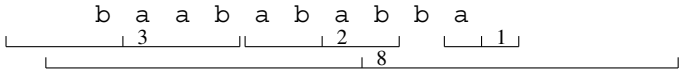
41 Critical Position of a Word

The existence of critical positions on a word is a wonderful tool for combinatorial analyses and also for the design of text algorithms. One of its striking application is the two-way string matching implemented in some C libraries.

Let x be a non-empty word and $\ell per(i)$ denote the ***local period*** at i on x, $i = 0, \ldots, |x|$, length of the shortest non-empty word w that is

$$\text{suffix of } A^*x[0\,..\,i-1] \text{ and } \text{prefix of } x[i\,..\,|x|-1]A^*.$$

In lay terms this roughly means the shortest non-empty square ww centred at position i has period $\ell per(i)$. For the word `baababababba` of period 8 we have $\ell per(1) = |\texttt{aab}| = 3$, $\ell per(6) = |\texttt{ab}| = 2$, $\ell per(10) = |\texttt{a}| = 1$ and $\ell per(7) = |\texttt{bbaababa}| = 8$. Some squares overflow the word to the right, to the left or to both.



Note that $\ell per(i) \le per(x)$ for any i. If $\ell per(i) = per(x)$, i is called a ***critical position***. When $x = u \cdot v$ its factorisation is said to be critical if $\ell per(|u|) = per(x)$.

Let MaxSuffix$(\le, x)$ and $ms =$ MaxSuffixPos$(\le, x)$ be the respective greatest suffix of x and its position according to the alphabet ordering $\le$.

Question. Let $x = yz$ where $z =$ MaxSuffix$(\le, x)$. Show that $|y| < per(x)$, that is MaxSuffixPos$(\le, x) < per(x)$.

Algorithm CriticalPos computes a critical position in linear time and constant extra space following Problems 38 and 40.

```
CriticalPos(x non-empty word)
1  i ← MaxSuffixPos(≤, x)
2  j ← MaxSuffixPos(≤^{-1}, x)
3  return max{i, j}
```

Applied to $x =$ `baababababba`, MaxSuffixPos$(\le, x) = 7$ is a critical position and, for this example, MaxSuffixPos$(\le^{-1}, x) = 1$ is not.

Question. Show that Algorithm CriticalPos computes a critical position on its non-empty input word.

https://doi.org/10.1017/9781108835831.004 Published online by Cambridge University Press

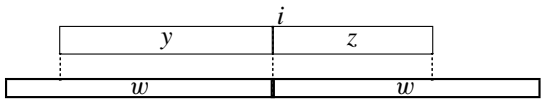
[**Hint:** Note the intersection of the two word orderings is the prefix ordering and use the duality between borders and periods.]

Solution

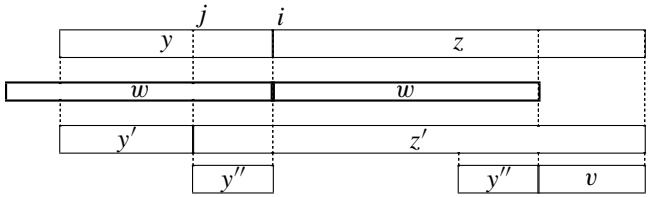
Answer to the first question. By contradiction assume $y \geq per(x)$ and let w be the suffix of y of length $per(x)$. Due to the periodicity, either w is a prefix of z or z is a suffix of w.
Case 1: If $z = ww'$, by its definition $www' < ww'$, then $ww' < w'$, a contradiction with the same definition.
Case 2: If $w = zz'$, $zz'z$ is a suffix of x, then $z < zz'z$, a contradiction with the definition of z.

Answer to the second question. The case of the alphabet of x reduced to one letter is easily solved because any position is critical. Then we consider both that x contains at least two different letters and w.l.o.g. that $i > j$ and show that i is a critical position.

Let w be the shortest non-empty square centred at i, that is, $|w| = \ell per(i)$. From the first question we know that y is a proper suffix of w.



Case 1: Assume z is a prefix of w. Then $|w| = per(x)$ because x being factor of ww has period $|w| = \ell per(i)$ that cannot be larger than $per(x)$. Thus i is a critical position.



Case 2: Assume $z = wv$ for a non-empty word v. Then $v < z$ by definition of z. Let $x = y'z'$, where $z' = \text{MaxSuffix}(\leq^{-1}, x)$ and y'' the non-empty word for which $y = y'y''$ and $z' = y''z$. Since $y''v$ is a suffix of x, it is smaller than $z' = y''z$ according to the ordering induced by $\leq^{-1}$, and thus v is smaller than z according to the same ordering. Referring to the hint, v being smaller than z according to both orderings is a prefix of z, then a border of it. Therefore $|w|$ is a period of z and consequently also of $x = yz$. Then as in the previous case i is a critical position.

This concludes the proof.

https://doi.org/10.1017/9781108835831.004 Published online by Cambridge University Press

Notes

The Critical Factorisation Theorem, the existence of a critical position on any non-empty word, is due to Cesari, Duval and Vincent [49, 105] (see Lothaire [175, chapter 8]).

The present proof appears in [96, 98] and is by Crochemore and Perrin [94], where it is part of the design of the two-way string-matching algorithm, which is time–space optimal. It is extended to a real-time algorithm by Breslauer et al. in [45].

42 Periods of Lyndon Word Prefixes

A Lyndon word is a non-empty self-minimal word, that is, it is alphabetically smaller than all its non-empty proper suffixes. The dual self-maximal words share some of their features. Lyndon words have useful properties for the design of matching algorithms and for the analysis of methods such as testing the cyclic equivalence of two words (Problem 37). The present problem deals with a remarkable simple solution to compute their prefix periods.

Let *period* denote the table of periods of prefixes of the word x. It is defined on non-empty prefix lengths ℓ, $\ell = 1, \ldots, |x|$ by

$$period[\ell] = \text{ smallest period of } x[0\,..\,\ell - 1].$$

For the word $x =$ `aababababba` we get

i	0	1	2	3	4	5	6	7	8
$x[i]$	a	a	b	a	b	a	b	b	a
ℓ	1	2	3	4	5	6	7	8	9
$period[\ell]$	1	1	3	3	5	5	7	8	8

Question. Show that Algorithm PrefixPeriods correctly computes the table of periods of prefixes of a Lyndon word.

https://doi.org/10.1017/9781108835831.004 Published online by Cambridge University Press

```
PrefixPeriods(x Lyndon word)
1  period[1] ← 1
2  p ← 1
3  for ℓ ← 2 to |x| do
4      if x[ℓ − 1] ≠ x[ℓ − 1 − p] then
5          p ← ℓ
6      period[ℓ] ← p
7  return period
```

Question. What changes are to be made to PrefixPeriods to compute the prefix periods of a self-maximal word?

Question. Show that testing if a word is a Lyndon word can be done in linear time with only constant extra space.

[**Hint:** Tune Algorithm PrefixPeriods.]

Solution

The solutions are very similar to those of Problem 39 although the notions of self-maximality and self-minimality are not strictly symmetric.

Adapting proofs in Problem 39, it is rather straightforward to prove that non-empty prefixes of a Lyndon word are of the form $u^e v$, where u is a Lyndon word and v is a proper prefix of u.

Correctness of PrefixPeriods. In Algorithm PrefixPeriods the variable p stores the period of the prefix $x[0\,..\,\ell-2]$. As recalled, the prefix is of the form $u^e v$ with $p = |u|$. The variable p is updated within the for loop. The comparison at line 4 is to check if the periodicity p continues, in which case p is also the period of $x[0\,..\,\ell-1]$ as done by the assignment at line 6. If the comparison is negative, we cannot have $x[\ell-1] < x[\ell-1-p]$ because the suffix $vx[\ell-1-p]$ would be smaller than x, a contradiction with the fact that x is a Lyndon word. Thus, in that case, $x[\ell-1] > x[\ell-1-p]$; this is the key feature mentioned in the solutions of Problem 39 and for which $x[0\,..\,\ell-1]$ is border free, then of period ℓ, as done by the assignment at line 5.

Periods of self-maximal prefixes. When the input to PrefixPeriods is self-maximal, without any change the algorithm computes its table of prefixes periods. The above argument is still valid after exchanging '<' and '>' essentially because the key feature also applies.

https://doi.org/10.1017/9781108835831.004 Published online by Cambridge University Press

Lyndon test. Algorithm LYNDON adapts the above algorithm and is like Algorithm SELFMAXIMAL (Problem 39) after exchanging '<' and '>'. An extra check is required to verify the whole word is border free.

```
LYNDON(x non-empty word)
1  p ← 1
2  for ℓ ← 2 to |x| do
3      if x[ℓ − 1] < x[ℓ − 1 − p] then
4          return FALSE
5      elseif x[ℓ − 1] > x[ℓ − 1 − p] then
6          p ← ℓ
7  if p = |x| then
8      return TRUE
9  else return FALSE
```

Prefixes of lengths 1, 3, 5, 7 and 8 of `aababababba` are Lyndon words. The word itself is not because it has border `a`.

At line 7, if $|x|$ is a multiple of p, x is a necklace; otherwise it is a prefix of a necklace.

43 Searching Zimin Words

The problem considers patterns that are words with variables. Besides the alphabet $\mathtt{A} = \{\mathtt{a}, \mathtt{b}, \ldots\}$ of constant letters, variables are from the (disjoint) alphabet $\mathtt{V} = \{\alpha_1, \alpha_2, \ldots\}$.

A pattern $P \in \mathtt{V}^*$ is said to match a word $w \in \mathtt{A}^*$ if $w = \psi(P)$, where $\psi : alph(P)^+ \to \mathtt{A}^+$ is a morphism. ***Zimin words*** Z_n, $n \geq 0$, play a crucial role in pattern avoidability questions (see Problem 93). They are defined by

$$Z_0 = \varepsilon \text{ and } Z_n = Z_{n-1} \cdot \alpha_n \cdot Z_{n-1}.$$

For example, $Z_1 = \alpha_1$, $Z_2 = \alpha_1\alpha_2\alpha_1$ and $Z_3 = \alpha_1\alpha_2\alpha_1\alpha_3\alpha_1\alpha_2\alpha_1$.

https://doi.org/10.1017/9781108835831.004 Published online by Cambridge University Press

The ***Zimin type*** of a word w is the greatest natural integer k for which $w = \psi(Z_k)$, where ψ is some morphism. The type is always defined since the empty word has type 0 and the type of a non-empty word is at least 1. For example, the Zimin type of $w = \texttt{adbadccccadbad}$ is 3 because it is the image of Z_3 by the morphism ψ defined by

$$\begin{cases} \psi(\alpha_1) = \texttt{ad}, \\ \psi(\alpha_2) = \texttt{b}, \\ \psi(\alpha_3) = \texttt{cccc}. \end{cases}$$

Question. Show how to compute in linear time Zimin types of all prefixes of a given word.

[**Hint:** Consider short borders of prefixes.]

Question. Show how to check in quadratic time if a given Zimin pattern occurs in a word.

[**Hint:** Consider Zimin types of words.]

Solution

Computing Zimin types. The computation of Zimin types of prefixes of $w \in \texttt{A}^+$ is done online on w as follows. Let $Ztype[i]$ be the type of the prefix of length i if w. We have $Ztype[0] = 0$. For other values, it is enough to prove they are computed iteratively via the equality

$$Ztype[i] = Ztype[j] + 1,$$

where $j = |ShortBorder(w[0\,..\,i-1])|$.

Letting $z = w[0\,..\,i-1]$ and $u = ShortBorder(z)$ we have $z = uvu$ for two words u and v, $v \neq \varepsilon$. By definition of $Ztype[j]$ the word u is the image of $Z_{Ztype[j]}$ by a morphism $\psi : \{\alpha_1, \alpha_2, \ldots, \alpha_{Ztype[j]}\}^+ \to \texttt{A}^+$. Extending the morphism by setting $\psi(\alpha_{Ztype[j]} + 1) = v$ shows that the Zimin type of z is at least $Ztype[j] + 1$. It remains to show that no border of z shorter than u can give a higher value.

Let $u'v'u'$ be a factorisation of z for which $Ztype[|u'|] = Ztype[|z|] - 1$ and assume by contradiction that u' is shorter than u with $Ztype[|u'|] > Ztype[|u|]$. Since then $Ztype[|u|] > 0$, $Ztype[|u'|] > 1$ which implies that $u' = u''v''u''$ for words such that $v'' \neq \varepsilon$ and $Ztype[|u''|] = Ztype[|u'|] - 1$. Then $Ztype[|u''|] = Ztype[|z|] - 2$. But since u'' is also a border of z, $Ztype[|z|] = Ztype[|u''|] + 1$, a contradiction.

https://doi.org/10.1017/9781108835831.004 Published online by Cambridge University Press

The algorithm computing short borders of prefixes (Problem 21) infers the solution.

Matching a Zimin pattern. The following fact reduces the question to the computation of Zimin types.

Fact. The prefix $w[0\,..\,i-1]$ of w matches a Zimin pattern Z_k if and only if $Ztype[i] \geq k$.

Indeed if a word is a morphic image of Z_j for some $j \geq k$ it is also a morphic image of Z_k.

The solution comes readily. The table *Ztype* is computed for each suffix z of w, which allows to detect prefixes of z that match Z_k. And if Z_k occurs in w it will be detected that way.

```
MATCHINGZIMINPATTERN(w non-empty word, k positive integer)
1  for s ← 0 to |w| − 1 do
2      Ztype[0] ← 0
3      for i ← s to |w| − 1 do
4          compute Ztype[i − s + 1] on w[s . . |w| − 1]
5          if Ztype[i − s + 1] ≥ k then
6              return TRUE
7  return FALSE
```

The computation at line 4 uses the linear-time algorithm from the previous question. Therefore the whole test takes $O(|w|^2)$ running time as expected.

Note that a simple modification of the algorithm can produce the largest integer k for which Z_k occurs in the input word.

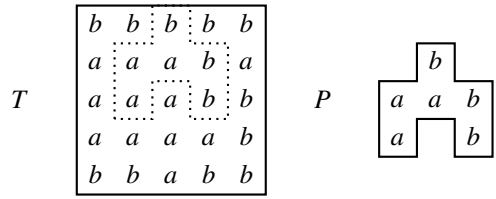
Notes

A more interesting question is the reverse pattern matching with variables, that is, to check if a given word with variables occurs in a given Zimin pattern. The problem is known to be in the NP class of complexity, but it is not known if it belongs to the NP-hard class.

https://doi.org/10.1017/9781108835831.004 Published online by Cambridge University Press

44 Searching Irregular 2D Patterns

Let P be a given (potentially) irregular two-dimensional (2D) pattern. By *irregular* is meant that P is not necessarily a rectangle, it can be of any shape. The aim is to find all occurrences of P in a 2D $n \times n'$ text T of total size $N = nn'$.



The figure shows an irregular pattern P. It fits in a 3×3 box and has 2 occurrences in T, one of them is shown.

Question. Show that two-dimensional pattern matching of irregular 2D-patterns can be done in $O(N \log N)$ time.

Solution

The solution is to linearise the problem. Let P be a non-rectangular pattern that fits into an $m \times m'$ box. Assume w.l.o.g. that the first and last column, as well as the first and last row of this box, contain an element of P. Otherwise rows or columns are removed.

Text T is linearised into T' by concatenating its rows. The transformation of P is more subtle. First P is inserted into the $m \times m'$ box whose elements that are not of P (empty slots) are changed to $*$. Rows of this box are concatenated, inserting between the rows the word of $n' - m'$ symbols $*$. This way P is linearised to P'.

Example. For P in the figure, rows of the 3×3 box are $*\,b\,*$, $a\,a\,b$, and $a\,*\,b$, and $P' = *\,b\,*\,*\,*\,a\,a\,b\,*\,*\,a\,*\,b$.

The basic property of the transformation is that occurrences of P in T correspond to occurrences of word P' in word T', where P' contains don't care symbols.

Consequently all occurrences of P can be found using the method of Problem 36. The total running time of the solution then becomes $O(N \log N)$.

Notes

The running time can be reduced to $O(N \log(\max(m, m')))$. The linearisation method presented here is used in [10].

https://doi.org/10.1017/9781108835831.004 Published online by Cambridge University Press