String Processing

- Typical applications:
  - pattern matching/recognition
  - molecular biology, comparative genomics, …
  - information retrieval
  - data/text mining
  - data/text compression, coding, encryption
  - string processing in large databases
  - …

Strings

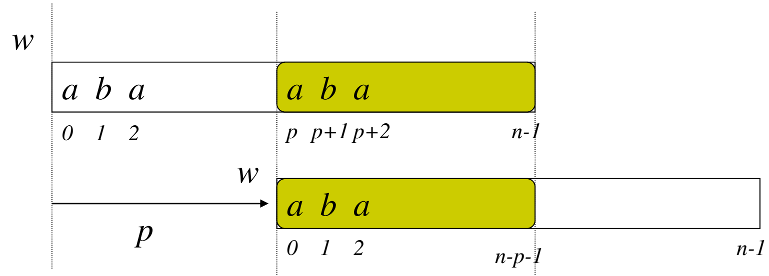
- A string is a *sequence of symbols* drawn from some well defined set call *the alphabet.*
- Examples of alphabets include:
  - ASCII code, Unicode
  - binary alphabet {0,1}
  - System of DNA base-pairs {A,C,G,T}
  - Latin, Greek, Chinese alphabet
- Examples of strings
  - Java/C/ADA programs, HTML/XML documents,…
  - DNA sequences, image/video/audio files

Strings

- Basic definitions:
  - Let A be an alphabet. We say that A^+ contains all non-empty strings based on symbols from A, and $A^*=A^+\cup\{\varepsilon\}$, where ε is an *empty string*.
  - Let w be a string of length n. We say that $w=w[0..n-1]$.
  - Any initial fragment $w[0..i]$ is called a *prefix* of w.
  - Any final fragment $w[j..n-1]$ is called a *suffix* of w.
  - Any fragment of form $w[i..j]$ is called a *substring* of w.

Periodicity

- We say that $w=w[0..n-1]$ has a *period* p **iff** $w[i]=w[i+p]$, for all $0\le i\le n-p-1$, see below



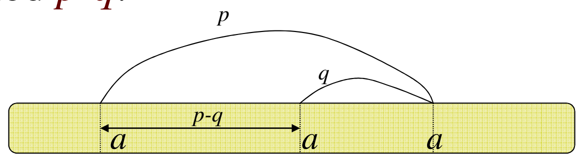
- For example string $w=abaabaabaaba$ has period 3

Periodicity Lemma

- But *w=abaabaabaaba* has also periods 6, 9 and 11
- ***Lemma:*** *p* is the shortest period in *w=w[0..n-1]* **iff** *w[0..n-p-1]* is the longest prefix of w which is also a suffix *w[p..n-1]*, see figure on previous slide
- ***Periodicty Lemma:*** If string *w=w[0..n-1]* has periods *p* and *q* such that $p+q \leq n$ then *w* has also period *gcd(p,q)*, where gcd(,) stands for the greatest common divisor of two integers.

Periodicity Lemma

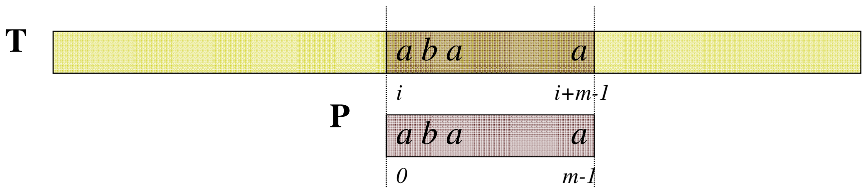
- ***Proof:*** The main observation is based on the fact that if string *w* has two periods $p \geq q$ then it also has period *p-q*.



- The thesis of the periodicity lemma follows from the observation (Euclid's Algorithm) that for any positive integers *a>b*, *gcd(a,b)=gcd(b,a-b)*.

String pattern matching

- ***Input:*** given two strings: *P=P[0..m-1]* called the *pattern* and *T=T[0..n-1]* called the *text*.
- ***Task:*** is to find *all occurrences* of *P* in *T* using as small number as possible text symbol comparisons, where an occurrence of *P* at position *i* in *T* is defined as *P[j]=T[i+j]* for all $0 \leq j \leq m-1$, see example below



Brute-Force Algorithm

- The brute-force algorithm tests naively (via consecutive symbols comparison) whether pattern *P* occurs at any permissible position $0 \leq i \leq n-m-1$ in text *T*.
- The test at each position can cost as much as m, for example when *T=aaaaaaaa..a* and *P=aaaaa*
  - possible scenario in images, unlikely in natural languages, codes
- Thus the time complexity of brute force algorithms is bounded by $(n-m) \cdot m = O(n \cdot m)$

Brute-force algorithm - code

- Algorithm *Brute-Force-First-Match(T,P)*: integer;

```
for i ← 0 to n-m-1 do
    j ← 0;
    while (j<m) and (T[i+j]=P[j]) do
        j ← j +1;
    if (j=m)
        then return (i);
return (-1);
```

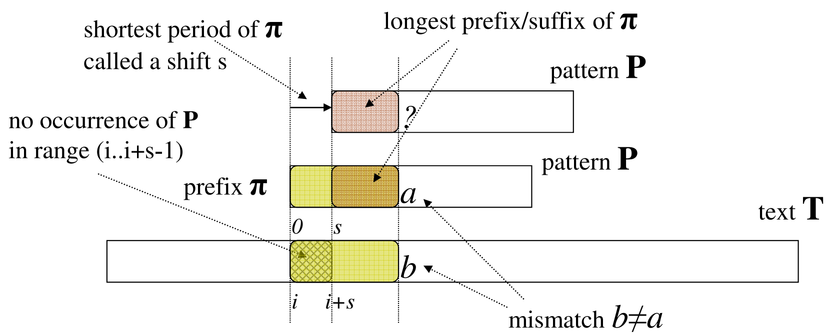

More efficient pattern matching

- Can we perform pattern matching in time $O(m+n)$ in any, even the worst case, scenario?
- The answer is yes, and the solution is based on proper use of periodicity of strings.
- But what is the cause of high complexity anyway?
- It must be multiple comparisons of text symbols.
- Can we do something about it?
- Indeed, we can, at least on most of the occasions.

Principle of Knuth-Morris-Pratt KMP Algorithm

- In Brute-force solution, when the algorithm moves from position i to $i+1$ it forgets all text symbols that have been recognized previously
- KMP algorithm similarly to Brute-force solution searches consecutive text positions storing at any time the longest currently recognized prefix π of P
- But when the mismatch between P and T is found KMP moves by the length of the smallest period of π remembering all recognized text symbols

Principle of Knuth-Morris-Pratt KMP Algorithm



- If a shorter than s shift was feasible s would not be the shortest period of π.

KMP Failure Function

- The KMP algorithm works in two stages: *pattern preprocessing* and actual *text search*.
- During pattern preprocessing we:
  - compute the longest proper prefix/suffix of each prefix *P[0..i]* and store its length in an array *F[1..m]* at position *i+1*. Vector *F* called the KMP *failure function*.
- During the text search we:
  - traverse consecutive text positions looking for pattern occurrences and avoiding redundant positive tests with a help of the failure function *F[1…m]*.

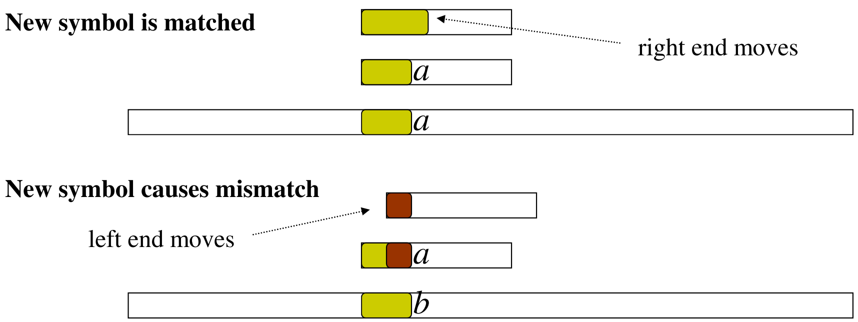
KMP failure function - example

- Let *P[0..5] = abaaba*
- Then the KMP failure function looks as follows
  - *F[0]* is not defined
  - *F[1] = 0* (string *a* has no proper prefix/suffix)
  - *F[2] = 0* (string *ab* has no proper prefix/suffix)
  - *F[3] = 1* (the longest prefix/suffix in *aba* is *a*)
  - *F[4] = 1* (the longest prefix/suffix in *abaa* is *a*)
  - *F[5] = 2* (the longest prefix/suffix in *abaab* is *ab*)
  - *F[6] = 3* (the longest prefix/suffix in *abaaba* is *aba*)

KMP algorithm - text search

```
Algorithm KMP-First-Match(T,P): integer;
i ← j ← 0;
while (j<m) and (T[i+j]=P[j]) {   //test next text symbol//
    j ← j +1;
    if (j=m) {
        then return (i);   // return the first occurrence of P //
        else if (j>0) {
                then { i ← i +(j-F[j]);  j ← F[j]; }  // shift based on F //
                else i+1;    // shift based on empty prefix //
        }
    }
    if (i>n-m) return (-1);   // end of the text, no pattern occurrences //
}
```


KMP text search complexity



- Since either left end of the recognized pattern prefix or its right end always move the time complexity (number of symbol comparisons) is bounded by *2n*.

KMP algorithm - preprocessing

- **Algorithm** *Brute-Force-KMP-Match(P)*: *integer*;

  $F[1] \leftarrow 0$;

  $i \leftarrow 1$; $j \leftarrow F[1]$;

  while ($i \leq m-1$) **do**

    if ($P[j]=P[i]$)

      then $F[i+1] \leftarrow j+1$; $j \leftarrow F[i+1]$; $i \leftarrow i+1$;

      else if ($j=0$)

        then $F[i+1] \leftarrow 0$; $j \leftarrow F[i+1]$; $i \leftarrow i+1$;

        else $j \leftarrow F[j]$;

KMP complexity

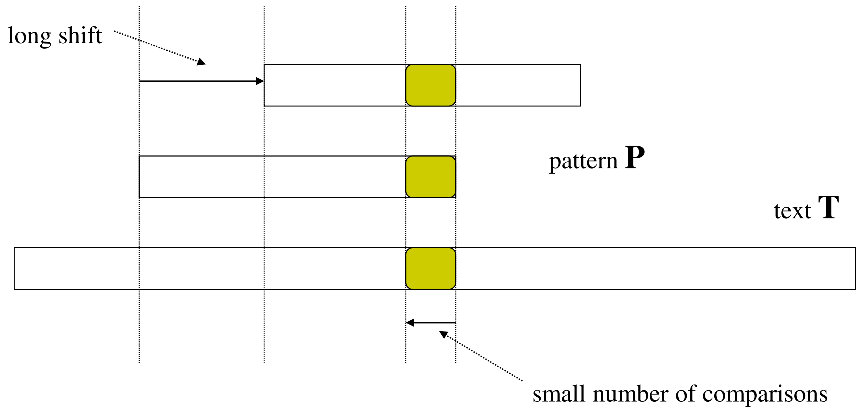
- Using similar argument to the one used in the text search one can prove that the preprocessing requires at most $2 \cdot m$ comparisons.
- ***Theorem:*** The total time (number of comparisons) complexity of KMP pattern matching algorithm is bounded by $2 \cdot m+2 \cdot n = O(m+n)$ and the extra space required for failure function is of size $O(m)$.
- We show later that one can obtain similar time bounds having only $O(1)$ space.

Other string matching algorithms

- *Boyer-Moore (BM) algorithm*
  - symbols in pattern P are tested against the text symbols from right to left, i.e., the algorithm is based on *suffix recognition*
  - this approach allows to perform text search in time $c \cdot n$, for constant $c<1$ on average (in random and natural texts), but the method works in time $O(n \cdot m)$ in the worst case.
  - It is possible to improve the worst time complexity of BM algorithm $O(n)$ if we keep in the memory information about the last recognized suffix of the pattern

Other string matching algorithms

- *Boyer-Moore algorithm*



Other string matching algorithms

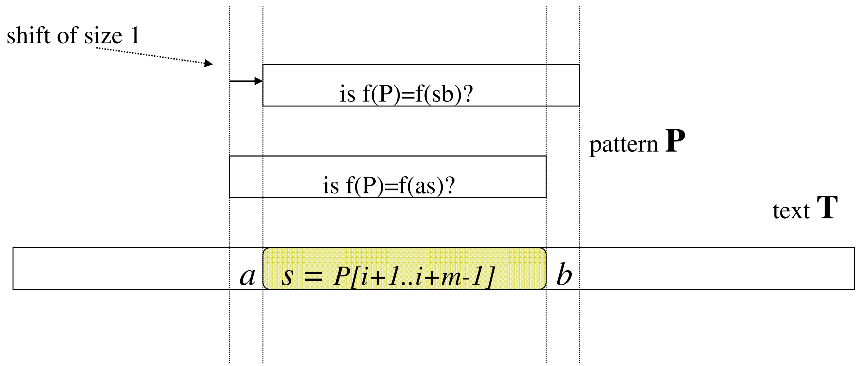
- *Karp-Rabin algorithm* is based on the use of a relatively simple *hash function* $f()$
  - each symbol a in the alphabet A has a unique integer score $s(a)$, e.g., all symbols can be enumerated from 1 to $|A|$ or using another (ASCII, Unicode) encoding
  - the score is extendable from symbols to strings with the help of a hash function $f()$, s.t.,
    - for $a,b \in A$ and strings s, $s_1=a\cdot s$, and $s_2=s\cdot b$
    - the score $f(s)$ is easily computable from $f(s_1)$ and $s(a)$, as well as
    - the score $f(s_2)$ is easily computable from $f(s)$ and $s(b)$

Other string matching algorithms

- *Karp-Rabin algorithm*
  - the algorithm computes initially the score $f(P)$
  - in the search stage it compares the score of consecutive text substrings $f(T[i..i+m-1])$, for all $i \in 1,\dots,n-m-1$
  - for every position i, s.t., $f(T[i..i+m-1])=f(P)$ we test the appropriate text and pattern symbols naively
- The algorithm works in time $O(n)$ on average (in random and natural texts) but in time $O(n\cdot m)$ in the worst case

Other string matching algorithms

- *Karp-Rabin algorithm*



Other string matching algorithms

- There exists an algorithm that uses $O(n\cdot log(m)/m)$ symbol comparisons in random texts after $O(m)$ time preprocessing; this is the best result possible in this model.
- There exists text search algorithm based on $n+O(n/m)$ symbol comparisons in the worst case after $O(m^2)$ time preprocessing; this is the best result possible in this model
- For extra information on string matching see:

  http://www-igm.univ-mlv.fr/~lecroq/string/index.html