Algorithms on Strings

Giulia Bernardini
giulia.bernardini@units.it

Fundamentals of algorithms
a.y. 2021/2022

Basic definitions

An alphabet Σ is a set of symbols (a.k.a. letters). It can be both finite or infinite: e.g., $\Sigma=\{0,1\}$, $\Sigma=\{a,\ldots,z\}$, $\Sigma=\mathbb{N}$ are all aceptable.

A string $S[1\ldots|S|]$ over Σ is a finite sequence of symbols from Σ; $|S|$ is the length of string S

Σ^* denotes the set of all possible strings over Σ; Σ^k denotes all possible strings of length k over Σ

ε denotes the empty string, which belongs to Σ^*

Basic definitions

If $x \in \Sigma^*$ and $y \in \Sigma^*$, then $xy \in \Sigma^*$ is their concatenation

Let $z = xyw$. Then:

- x is a prefix of z. If $x \neq z$ and $x \neq \varepsilon$, the prefix is proper
- w is a suffix of z. If $w \neq z$ and $w \neq \varepsilon$, the suffix is proper
- y is a substring of z. If $y \neq z$ and $y \neq \varepsilon$, the substring is proper

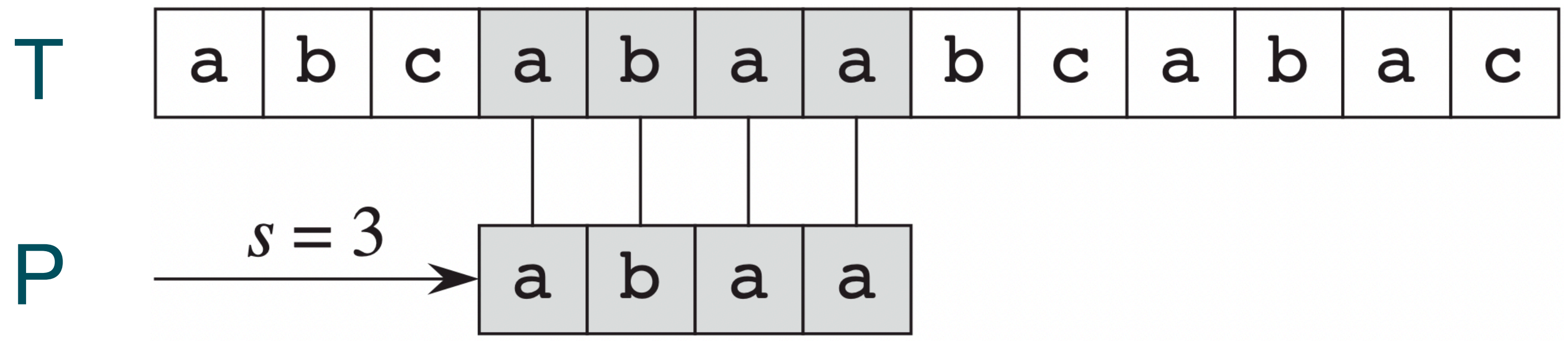
For any $1 \leq i \leq j \leq |S|$, $S[i..j]$ denotes the substring of length $j-i+1$ starting at position i of S

Basic definitions

Consider two strings, T[1..n] of length n and P[1..m] of length m≤n, both over the finite alphabet Σ.

P occurs with shift s (equivalently, occurs at position s+1) in T if 0 ≤ s ≤ n-m and T[s+1..s+m]=P[1..m].

If P occurs with shift s in T, then we call s a valid shift; otherwise, we call s an invalid shift.



We call text the longer string T; pattern the shorter string P

String-matching problem

The string-matching (a.k.a. pattern matching) problem is to find all positions at which a pattern P occurs in a text T.

This problem is the core of a myriad of applications involving text processing. Two prime areas in which algorithms on strings are fundamental are data mining and bioinformatics.

Applications: string sanitization

Data mining = extracting and discovering patterns in large data sets

It often raises privacy concerns: private and sensitive information can be mined from personal data, which often consists of strings: e.g., analysing a sequence of locations visited by an individual can expose visits to mental health clinics; a sequence of websites visited by an individual may expose sexual and political orientations…

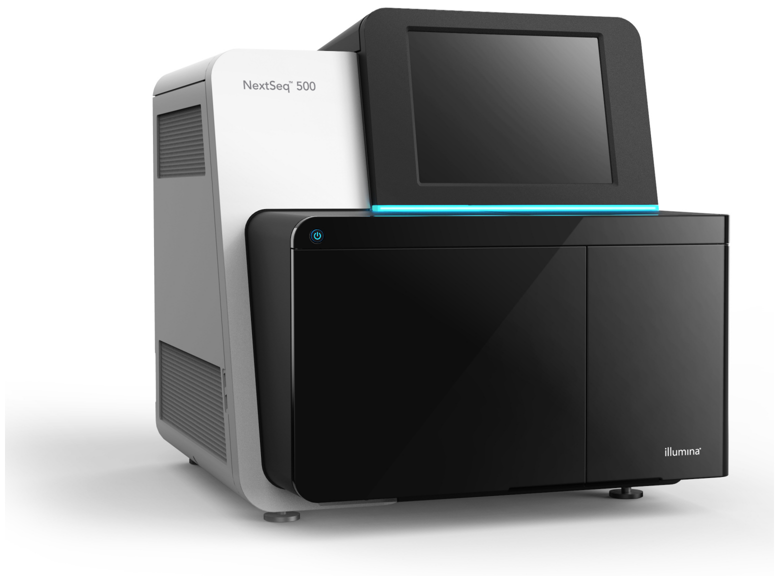
Possible solutions to these problems are string sanitization techniques: the sensitive information is encoded by a set of patterns, which need to be concealed from a string. These techniques often require efficient pattern patching.

Applications: genome analysis

Human DNA can be modeled as a pair of strings over an alphabet of four letters, called bases: {A,C,G,T}. Each of these strings has a length of about $3 \cdot 10^9$.

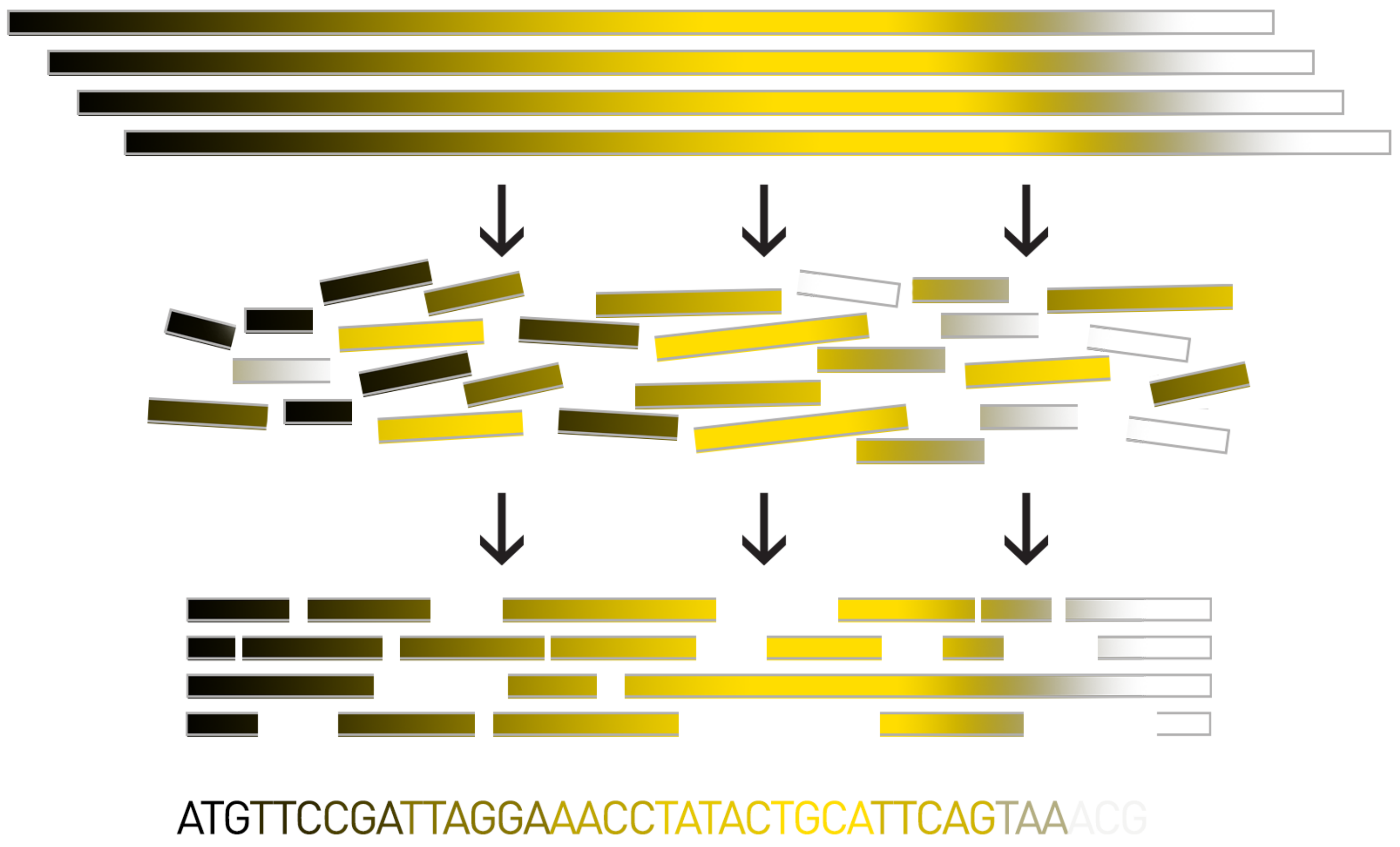
The bases of DNA can be read using sequencers. No existing sequencer can read all the bases of the genome in one go: they all produce a set of fragments, called reads. The most widely used sequencer, Illumina, produces reads of length aroud 100 bases.

Assembling the reads to reconstruct the whole genome is difficult…



Applications: genome analysis

Tecnologies to read fragments of the genome exist from the Seventies: the “first generation” sequencers produced reads of length about 1000 bases. But there existed no algorithms to arrange these reads to reconstruct the human genome.



Applications: genome analysis

The Human Genome Project was launched in 1990 with a budget of $3 billion and the objective of reconstructing the sequence of a whole human genome within 15 years.

A 'rough draft' of the genome was finished in 2000, especially thanks to the design of efficient string algorithms.
The first fairly complete version of the genome was released in 2003: it covered 92% of the genome with an accuracy over 99,99%.

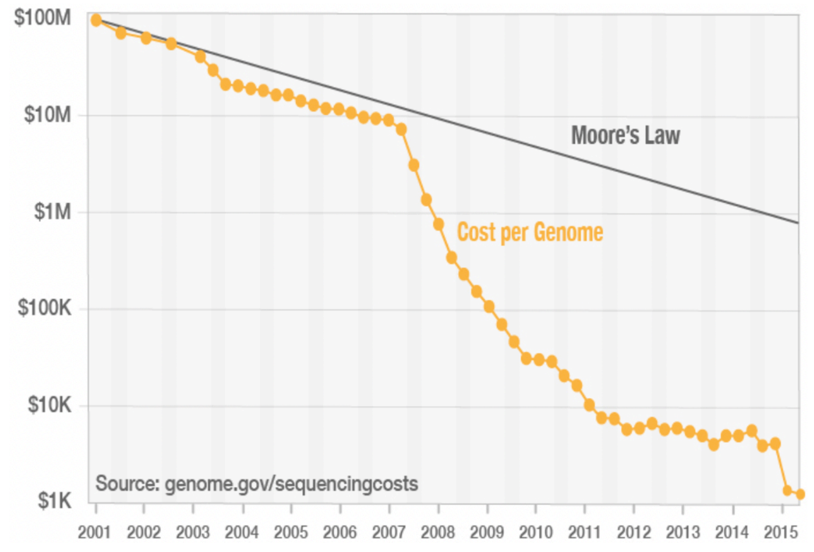
The last 8% of the genome was reconstructed in January 2022.

Applications: genome analysis

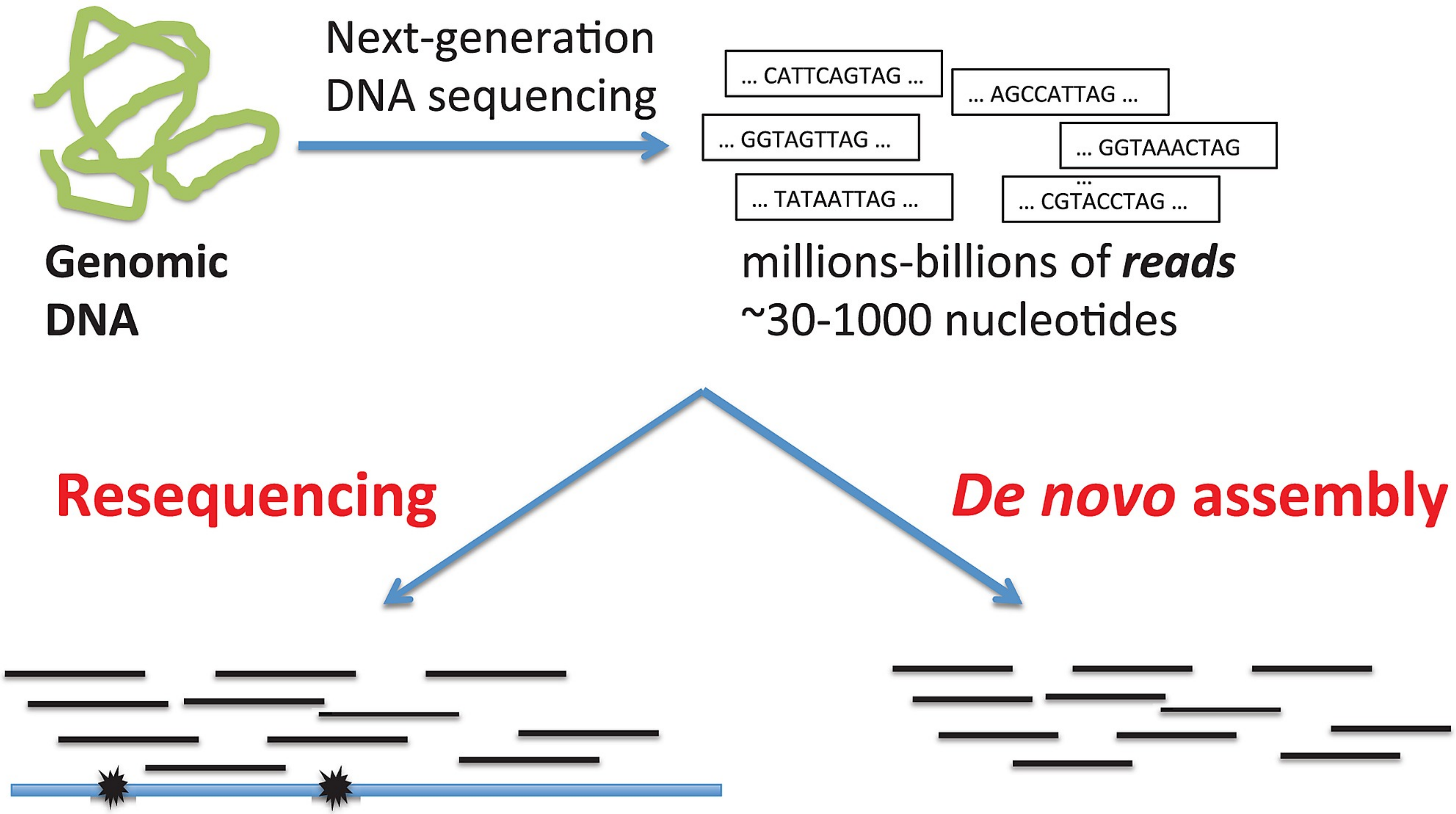
So “de novo” assembly (i.e., to assembly reads without a reference genome) is an extremely difficult task, and it costed billions of dollars.

Since 2003, all newly sequenced human genomes are assembled using the first sequenced genome as a reference. This task is much easier, and involves mapping the reads to the reference. This involves string-matching algorithms.

Sequencing technologies become faster and cheaper (sequencing a genome now costs around 1000$), thus this procedure has become very common.



Applications: genome analysis



Align reads to *reference genome* and identify variants

Construct genome sequence from overlaps between reads

Exact String Matching

Reference1: Part of chapter "String Matching" of: Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. *Introduction to algorithms*. (Chapters 32.1 and 32.4 of the third edition)
Reference2: Chapter 2.2 of: Gusfield, D. *Algorithms on Strings, Trees and Sequences.*

The string-matching problem

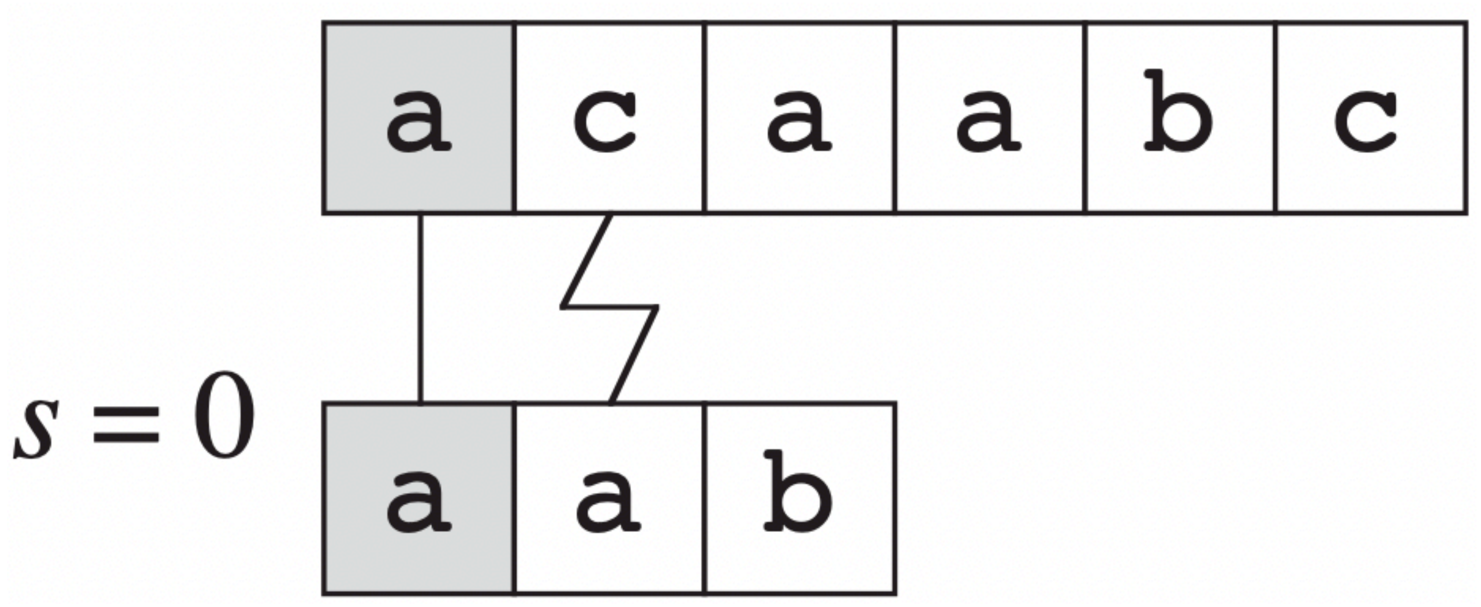
Input: a text T of length n and a pattern P of length m≤n

Output: all the occurrences of P in T

The string-matching problem

Input: a text T of length n and a pattern P of length m≤n

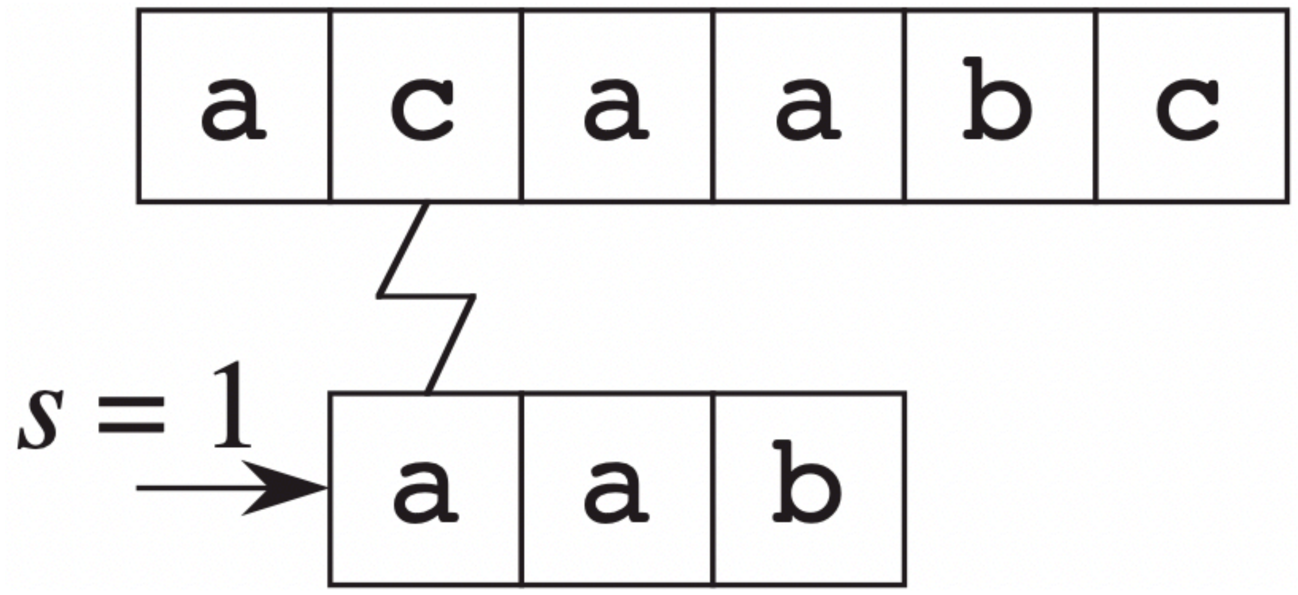
Output: all the occurrences (or valid shifts) of P in T



The string-matching problem

Input: a text T of length n and a pattern P of length m≤n

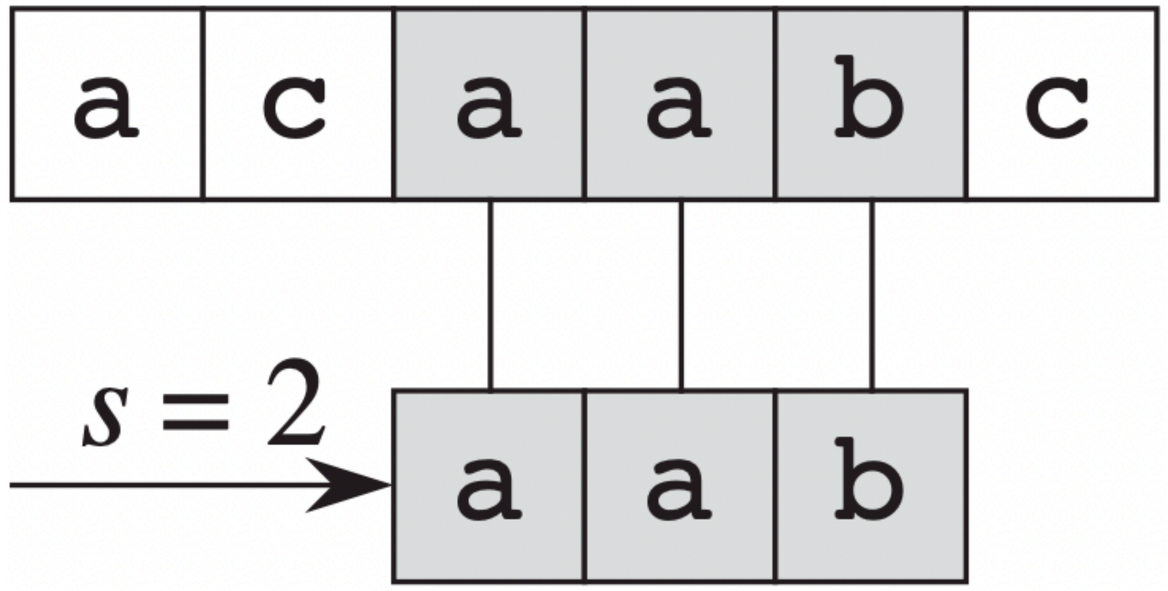
Output: all the occurrences (or valid shifts) of P in T



The string-matching problem

Input: a text T of length n and a pattern P of length m≤n

Output: all the occurrences (or valid shifts) of P in T

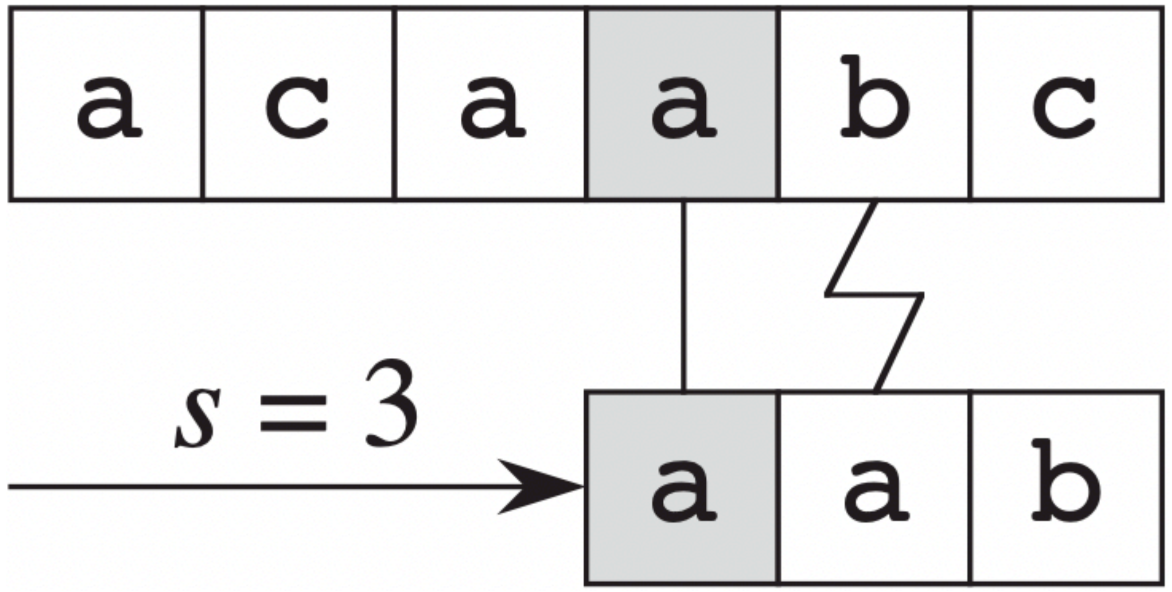


OUTPUT: shift 2 (or position 3)

The string-matching problem

Input: a text T of length n and a pattern P of length m≤n

Output: all the occurrences (or valid shifts) of P in T



OUTPUT: shift 2 (or position 3)

The string-matching problem

The naive solution (compare the letters of P starting from each possible position in T) requires *O*(nm) time.

```
NAIVE_STRING_MATCHING(T,P)
    sol←emptylist;
    **for** s=0 **to** |T|-|P|                          ┐
        i←1;                                            │
            **while** i≤|P| **and** T[s+i]=P[i]  ┐          │
                i←i+1;                        ┘ O(|P|)  │ O(|T|)
            **if** i>|P|                                  │
                sol.append(s);                          ┘
    **return** sol;
```

Towards a better solution

Our goal: designing an $O(|T|+|P|)$-time algorithm

General idea: to skip some comparisons (i.e., to make longer shifts) by first spending a few time on learning the internal structure of the pattern or the text (e.g., are there repeated substrings? Which characters are in P?). This part of an algorithm is called preprocessing.

Preprocessing the pattern

Given a position i in P, let π_i be the length of the longest proper suffix of P[1..i] that matches a prefix of P. For example:

P = ababaca

$\pi_3=1$; $\pi_4=2$; $\pi_5=3$; $\pi_7=1$

These values encapsulate knowledge about how the pattern matches against shifts of itself. If q characters of P have matched The text T at shift s, the next potentially valid shift is at $s'=s+(q-\pi_q)$.

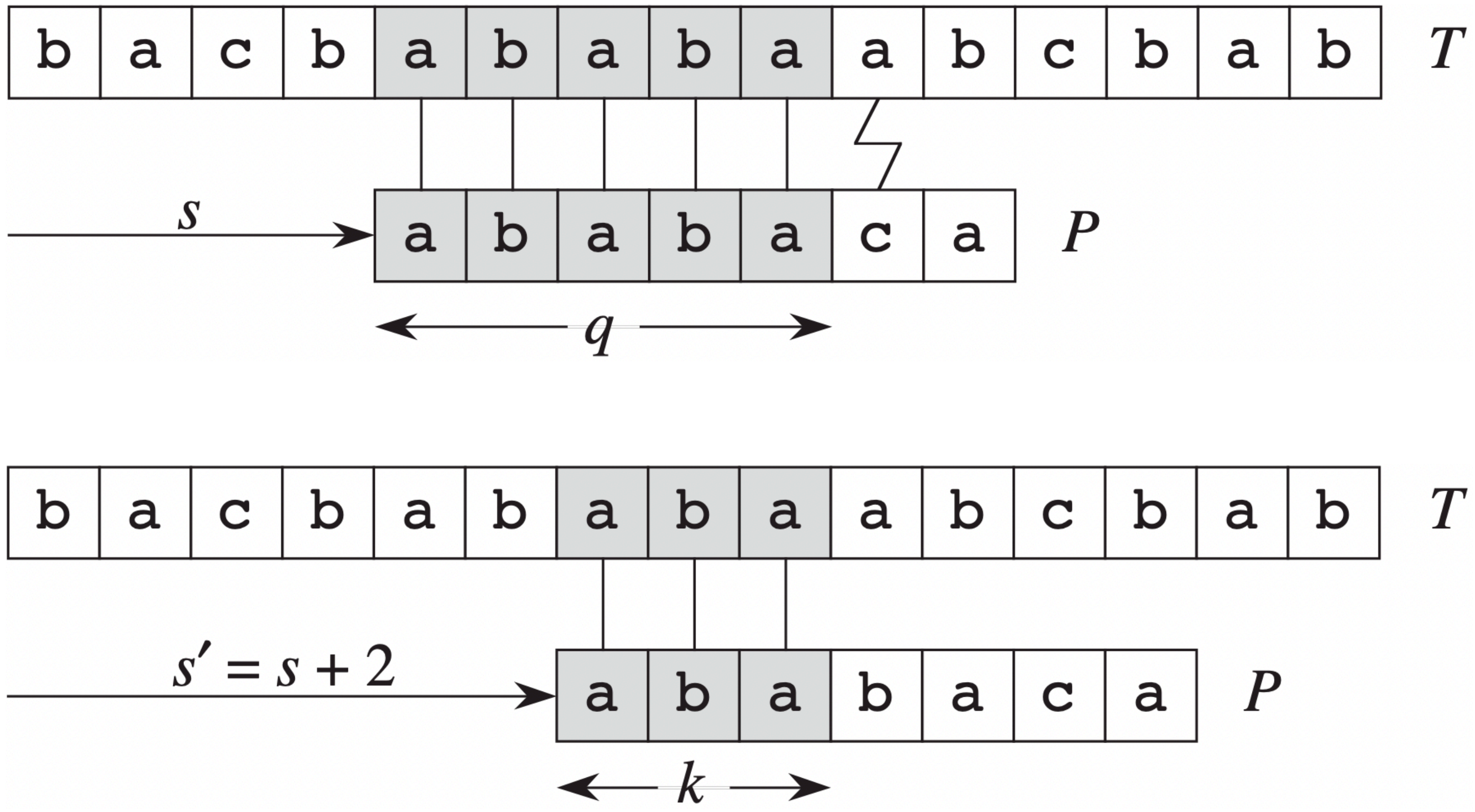
Preprocessing the pattern

Given a position i in P, let π_i be the length of the longest proper suffix of P[1..i] that matches a prefix of P. For example:

P = ababaca

$\pi_3=1$; $\pi_4=2$; $\pi_5=3$; $\pi_7=1$

These values encapsulate knowledge about how the pattern matches against shifts of itself. If q characters of P have matched The text T at shift s, the next potentially valid shift is at $s'=s+(q-\pi_q)$.

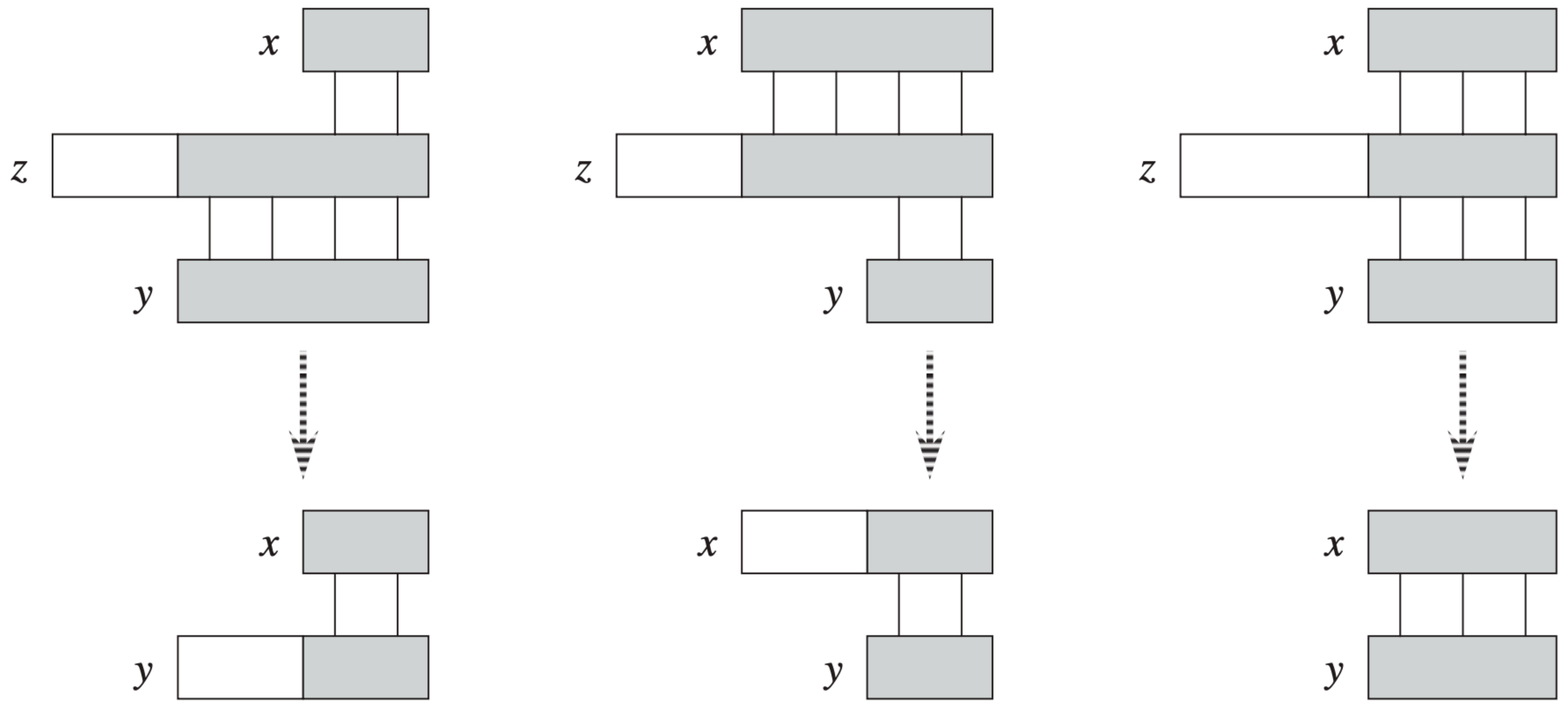


Preprocessing the pattern

Overlapping-suffix Lemma. Let x,y,z be strings such that both x and y are a suffix of z. Then, if (a) $|x|<|y|$, x is also a suffix of y; if (b) $|x|>|y|$, y is also a suffix of x; if (c) $|x|=|y|$, then $x=y$.

Preprocessing the pattern

Overlapping-suffix Lemma. Let x,y,z be strings such that both x and y are a suffix of z. Then, if (a) |x|<|y|, x is also a suffix of y; if (b) |x|>|y|, y is also a suffix of x; if (c) |x|=|y|, then x=y.



(a) (b) (c)

Preprocessing the pattern

```
COMPUTE_PREFIX_FUNCTION(P)
  π[1..|P|]←emptyarray;
  π[1]←0;
  k←0;
  for q=2 to |P|
    while k>0 and P[k+1]≠P[q]
      k←π[k];
    if P[k+1]=P[q]
      k←k+1;
    π[q]←k;
  return π;
```

Preprocessing the pattern

```
COMPUTE_PREFIX(P)
  1. π[1..|P|]←emptyarray;
  2. π[1]←0;
  3. k←0;
  4. for q=2 to |P|
     5. while k>0 and P[k+1]≠P[q]
        6. k←π[k];
     7. if P[k+1]=P[q]
        8. k←k+1;
     9. π[q]←k;
  10. return π;
```

k is increased at most once for each iteration of the for loop, thus the total increase of k is at most |P|-1.

Each time we enter the for loop, k<q; the for increments q, so k<q holds all the time. Then instructions 2. and 9. ensure that π[q]<q for all q and thus k is always decreased in the while loop.

π has no negative values, so k is never negative.

Preprocessing the pattern

```
COMPUTE_PREFIX(P)
  1. π[1..|P|]←emptyarray;
  2. π[1]←0;
  3. k←0;
  4. **for** q=2 **to** |P|
    5. **while** k>0 **and** P[k+1]≠P[q]
      6. k←π[k];
    7. **if** P[k+1]=P[q]
      8. k←k+1;
    9. π[q]←k;
  10. **return** π;
```

- increase of k is at most |P|-1
- k is always decreased in the while loop
- k is never negative

The total decrease in k from the while loop is bounded from above by the total increase in k over all iterations of the for loop, which is |P|-1.

The running time of COMPUTE_PREFIX(P) is thus $\Theta(|P|)$.

Preprocessing the pattern

Is COMPUTE_PREFIX a correct procedure?

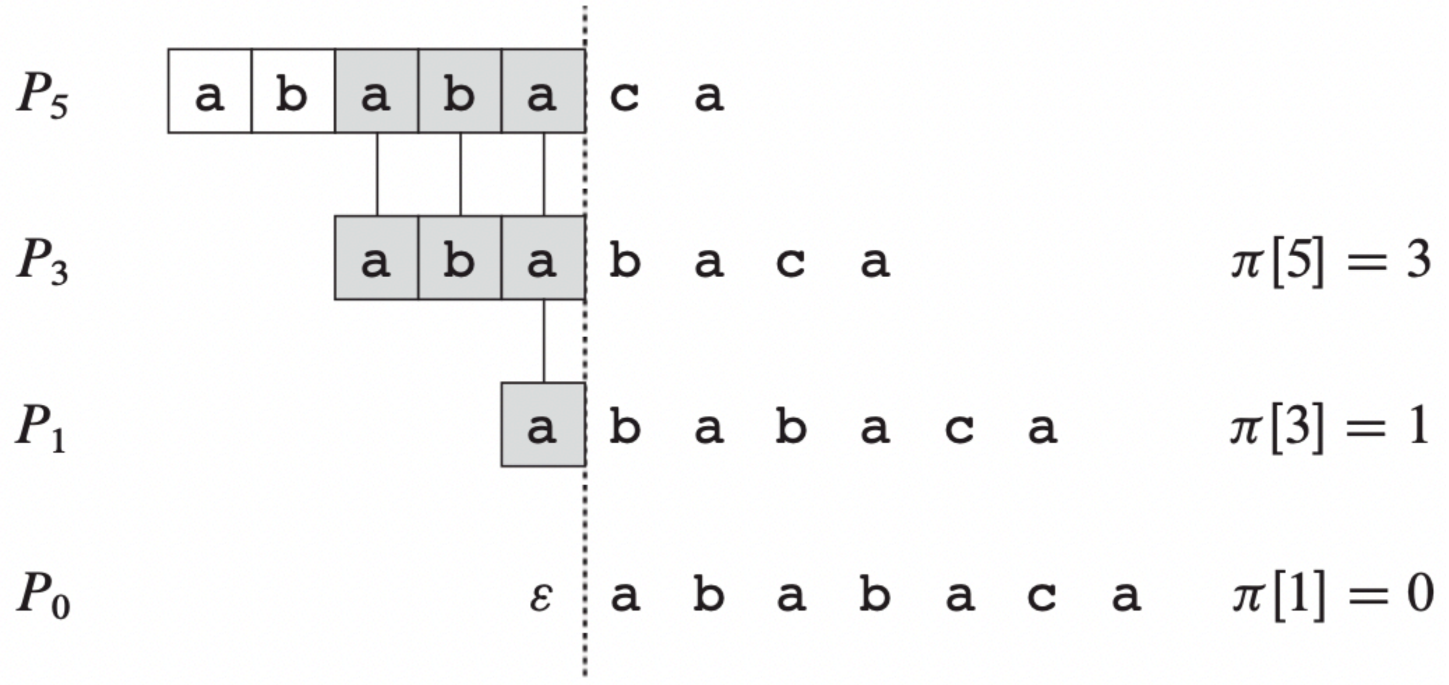
Let P_i denote the prefix P[1..i]. We formally define the prefix function $\pi[q]=\max\{k : k<q \text{ and } P_k \text{ is a suffix of } P_q\}$. The iterate prefix function is defined as $\pi^*[q]=\{\pi[q],\pi^{(2)}[q],\ldots,\pi^{(t)}[q]\}$, where $\pi^{(i)}[q]=\pi[\pi^{(i-1)}[q]]$ and the sequence stops when reaching $\pi^{(t)}[q]=0$.

Prefix-function iteration lemma. For any q=1,…,|P|, we have $\pi^*[q]=\{k : k<q \text{ and } P_k \text{ is a suffix of } P_q\}$.

In other words, $\pi^*[q]$ contains the lengths of all prefixes of P that match a suffix of P_q .

Here, $\pi^*[5]=\{3,1,0\}$

i	1	2	3	4	5	6	7
$P[i]$	a	b	a	b	a	c	a
$\pi[i]$	0	0	1	2	3	0	1



Preprocessing the pattern

Lemma. For q =1,2,…,|P|, if $\pi[q]>0$, then $\pi[q]-1\in \pi^*[q-1]$

Proof. Let $r=\pi[q]>0$, so that $r<q$ (because it is the length of a proper suffix of the prefix of P of length q). Thus $r-1<q-1$ and P_{r-1} is a proper suffix of P_{q-1} (by dropping the last character of P_r and P_q: see the example below).

By the Prefix-function iteration lemma, $r-1\in \pi^*[q-1]$. Thus we have that $\pi[q]-1 = r-1 \in \pi^*[q-1]$.

P = ababaca

P_5 = ababa

Preprocessing the pattern

Lemma. For q =1,2,…,|P|, if $\pi[q]>0$, then $\pi[q]-1 \in \pi^*[q-1]$

Proof. Let $r=\pi[q]>0$, so that $r<q$ (because it is the length of a proper suffix of the prefix of P of length q). Thus $r-1<q-1$ and P_{r-1} is a proper suffix of P_{q-1} (by dropping the last character of P_r and P_q: see the example below).

By the Prefix-function iteration lemma, $r-1 \in \pi^*[q-1]$. Thus we have that $\pi[q]-1 = r-1 \in \pi^*[q-1]$.

P = ababaca

P_5 = ababa $\qquad$ $r=\pi[5]=3$

Preprocessing the pattern

Lemma. For $q = 1,2,\ldots,|P|$, if $\pi[q]>0$, then $\pi[q]-1 \in \pi^*[q-1]$

Proof. Let $r=\pi[q]>0$, so that $r<q$ (because it is the length of a proper suffix of the prefix of P of length q). Thus $r-1<q-1$ and P_{r-1} is a proper suffix of P_{q-1} (by dropping the last character of P_r and P_q: see the example below).

By the Prefix-function iteration lemma, $r-1 \in \pi^*[q-1]$. Thus we have that $\pi[q]-1 = r-1 \in \pi^*[q-1]$.

P = ababaca	
P_5 = ababa	$r=\pi[5]=3$
P_4 = abab	$r-1=\pi[5]-1=2$
P_2 = ab	P_2 is equal to a suffix of $P_{q-1} = P_4$

Preprocessing the pattern

Lemma 1. For q =1,2,…,|P|, if $\pi[q]>0$, then $\pi[q]-1 \in \pi^*[q-1]$

Let $E_{q-1} = \{k \in \pi^*[q-1] : P[k+1]=P[q]\}$: these are all $k<q-1$ s.t. P_k is equal to a suffix of P_{q-1} and P_{k+1} is equal to a suffix of P_q. It holds the following corollary of Lemma 1.

$$\pi[q] = \begin{cases} 0 & \text{if } E_{q-1} = \emptyset \\ 1+\max\{k \in E_{q-1}\} & \text{otherwise} \end{cases}$$

We use this relation and the two lemmas to prove the correctness of COMPUTE_PREFIX.

Preprocessing the pattern

```
COMPUTE_PREFIX(P)
    1. $\pi[1..|P|] \leftarrow$ emptyarray;
    2. $\pi[1] \leftarrow 0$;
    3. $k \leftarrow 0$;
    4. **for** $q=2$ **to** $|P|$
        5. **while** $k>0$ **and** $P[k+1] \neq P[q]$
            6. $k \leftarrow \pi[k]$;
        7. **if** $P[k+1]=P[q]$
            8. $k \leftarrow k+1$;
        9. $\pi[q] \leftarrow k$;
    10. **return** $\pi$;
```

At the start of each iteration of the for loop we have $k=\pi[q-1]$ (by initialisation and line 9). Lines 5-8 adjust k so that it becomes the correct value of $\pi[q]$.

The while loop of lines 5–6 searches through all values $k \in \pi^*[q-1]$ until it finds a value of k for which $P[k+1]=P[q]$.

At that point, k is the largest value in the set E_{q-1}, so that we can set $\pi[q]$ to $k+1$.

Preprocessing the pattern

```
COMPUTE_PREFIX(P)
  1. $\pi[1..|P|] \leftarrow$ emptyarray;
  2. $\pi[1] \leftarrow 0$;
  3. $k \leftarrow 0$;
  4. for $q=2$ to $|P|$
    5. while $k>0$ and $P[k+1] \neq P[q]$
      6. $k \leftarrow \pi[k]$;
    7. if $P[k+1]=P[q]$
      8. $k \leftarrow k+1$;
    9. $\pi[q] \leftarrow k$;
  10. return $\pi$;
```

If the while loop cannot find a $k \in \pi^*[q-1]$ such that $P[k+1]=P[q]$, then k equals 0 at the end of the loop.

If $P[1]=P[q]$, then we should set both k and $\pi[q]$ to 1; otherwise we should leave k alone and set $\pi[q]$ to 0.

Lines 7–9 set k and $\pi[q]$ correctly in either case.

The Knuth-Morris-Pratt algorithm

The KMP algorithm exploits the prefix function to skip unnecessary shifts.

```
KMP(T,P)
  1. $\pi \leftarrow$COMPUTE_PREFIX(P);
  2. $q \leftarrow 0$;               //q stores the number of matched chars of P
  3. sol$\leftarrow$emptylist;
  4. for i = 1,…,|T|
      5. while $q>0$ and $P[q+1] \neq T[i]$
          6. $q \leftarrow \pi[q]$;                 //next character does not match
      7. if $P[q+1]=T[i]$
          8. $q \leftarrow q+1$;                    //next character matches
      9. if $q=|P|$
          10. sol.append(i-|P|)
          11. $q \leftarrow \pi[q]$;                 //look for the next match
```

The Knuth-Morris-Pratt algorithm

The time complexity of KMP is $\Theta(|P|+|T|)$. The analysis of the algorithm is entirely analogous to the one of COMPUTE_PREFIX.

```
KMP(T,P)
  1. π←COMPUTE_PREFIX(P);
  2. q←0;                    //q stores the number of matched chars of P
  3. sol←emptylist;
  4. for i = 1,…,|T|
      5. while q>0 and P[q+1]≠T[i]
          6. q←π[q];                    //next character does not match
      7. if P[q+1]=T[i]
          8. q←q+1;                     //next character matches
      9. if q=|P|
          10. sol.append(i-|P|)
          11. q←π[q];                   //look for the next match
```

The Boyer-Moore algorithm

Boyer-Moore is the practical method of choice for exact matching: it typically examines less than |P|+|T| characters, so it has an expectes sublinear running time and a linear worst-case time.

It uses three clever ideas:

1. The characters of the pattern are scanned from right to left
2. It uses the bad character shift rule
3. It uses the good suffix shift rule

Right-to-left scan

The characters of the pattern are scanned from right to left; then the pattern is shifted to the right, like in KMP and in the naive algorithm.

```
T= xpbctbxabpqxctbpq
           |
P=   tpabxab
```

Right-to-left scan

The characters of the pattern are scanned from right to left; then the pattern is shifted to the right, like in KMP and in the naive algorithm.

```
T= xpbctbxabpqxctbpq
          ||
P=   tpabxab
```

Right-to-left scan

The characters of the pattern are scanned from right to left; then the pattern is shifted to the right, like in KMP and in the naive algorithm.

```
T= xpbctbxabpqxctbpq
         |||
P=   tpabxab
```

Right-to-left scan

The characters of the pattern are scanned from right to left; then the pattern is shifted to the right, like in KMP and in the naive algorithm.

```
T= xpbctbxabpqxctbpq
        ||||
P=   tpabxab
```

Right-to-left scan

The characters of the pattern are scanned from right to left; then the pattern is shifted to the right, like in KMP and in the naive algorithm.

```
T= xpbctbxabpqxctbpq
       x||||
P=   tpabxab ──→
```

The good suffix rule

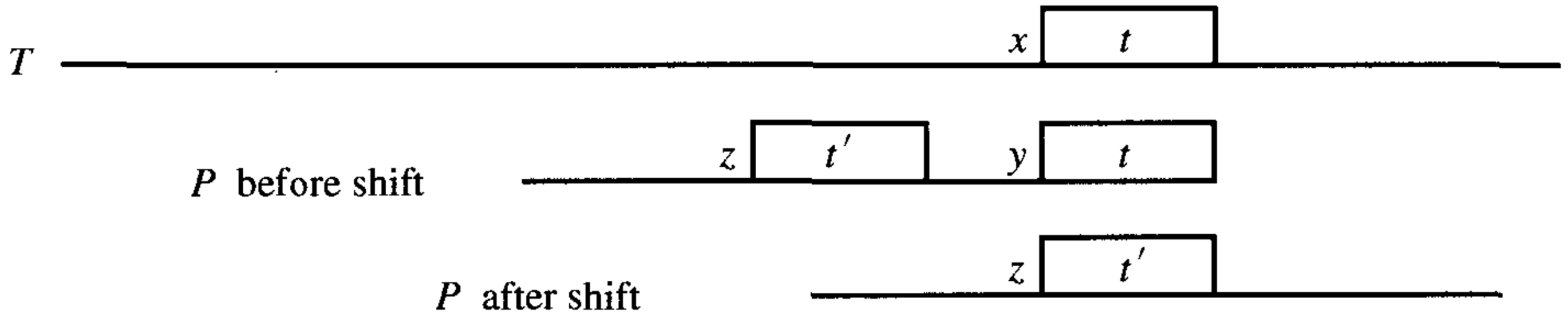
If t = P[i .. |P |] = T[i + j .. |P | + j] and P [i – 1] ≠ T [i + j – 1] then find, if it exists, the rightmost copy t' of t in P such that the character to the left of t' in P differs from the character to the left of t.

Shift P to the right so that t' is below t in T.

If t' does not exist, shift the left end of P past the left end of t in T by the least amount so that a prefix of P matches a suffix of t in T.

If no such shift is possible, then shift P |P| positions to the right.

If an occurrence of P is found, then shift P by the least amount so that a proper prefix of P matches a suffix of the occurrence of P.



The good suffix rule

If $t = P[i \,..\, |P|] = T[i + j \,..\, |P| + j]$ and $P[i - 1] \neq T[i + j - 1]$ then find, if it exists, the rightmost copy t’ of t in P such that the character to the left of t’ in P differs from the character to the left of t.

Shift P to the right so that t’ is below t in T.

If t’ does not exist, shift the left end of P past the left end of t in T by the least amount so that a prefix of P matches a suffix of t in T.

If no such shift is possible, then shift P |P| positions to the right.

If an occurrence of P is found, then shift P by the least amount so that a proper prefix of P matches a suffix of the occurrence of P.

Preprocessing P for the good suffix rule requires $O(|P|)$ time.

The bad character shift rule

Suppose that the last |P|-i characters of P matched the corresponding characters of T, up to position k+1 in T. If P[i]≠T[k],

```
T=babcabcaab
     x||
 P=cabab
```

The bad character shift rule

Suppose that the last |P|-i characters of P matched the corresponding characters of T, up to position k+1 in T. If P[i]≠T[k], let j be the rightmost occurrence in P of character T[k] (j=0 if it does not occur anywhere in P). If j<i, shift P i-j positions to the right, so that T[k] and P[j] are aligned (if j>0) or P starts after T[k] (if j=0).

If j>i, shift P one position to the right.

```
T=babcabcaab
      ¦||
 P=cabab
      ¦
   P=cabab
```

Preprocessing P for the bad character rule

Let Σ be the alphabet of T (note that we can assume $|\Sigma| \leq |T|$).

- Initialise an array of zeroes R of length $|\Sigma| \leq |T|$
- For each $i=1,\ldots,|P|$, $R[P[i]] \leftarrow i$
- At the end, R[x] contains the rightmost position of P where character x occurs; or 0 if x does not occur in P.
- This preprocessing requires $\Theta(|\Sigma|+|P|)$ time

The Boyer-Moore algorithm

- Start from the leftmost alignment of P in T and start matching the characters of P backwards (right-to-left)
- If a mismatch is found, select the largest shift between the one given by the good suffix rule and the one of the bad character rule
- If an occurrence of P is found, output the occurrence and shift P according to the good suffix rule.
- The worst-case running time of this algorithm is $\Theta(|P||T|)$, but it has expected sublinear time. By adding one extra rule, it is possible to obtain an $O(|P|+|T|)$-time algorithm.

The Boyer-Moore-Galil algorithm

When P does not occur in T (or it only has a constant number of occurrences), the running time of Boyer-Moore is *O*(|P|+|T|) time: the nasty case is when there are many occurrences of P.

The Galil rule addresses this case and makes it possible to achieve *O*(|T|+|P|) worst-case running time .

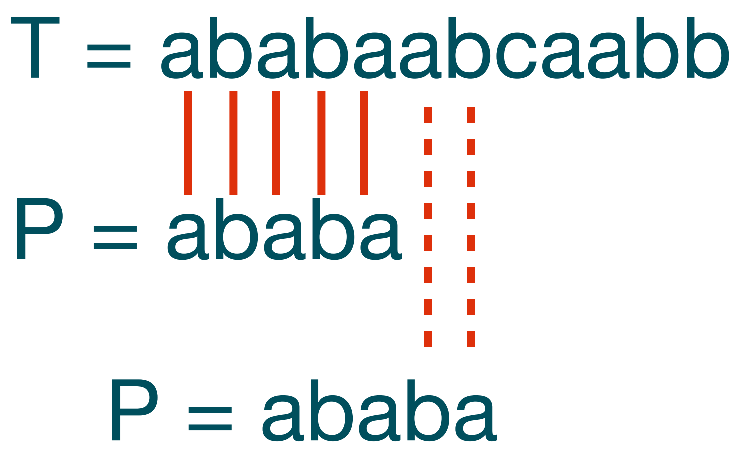
This rule is concerned with the comparisons that need to be done after an occurrence of P in T has been found. It exploits the possible repetitive nature of the pattern to skip useless comparisons after a full matching of P.

The period of a string S is the smallest p s.t. S[i]=S[i+p] for all i=1,…,|S|-p. E.g., the period of S=abababababa is p=2. The period of S=abcdab is p=4. The period of S=abcdefg is p=7.

The Boyer-Moore-Galil algorithm

The Galil rule is as follows.

Let k be the period of P. When a match of P has been found, shift it by k positions to the right. Then there is an occurrence at this new position of T if and only if the last k characters of P match the corresponding characters of T.



k=2

Only do the last two comparisons. Each time this rule is applied, it avoids |P|-k comparisons.

The Boyer-Moore-Galil algorithm

What would the good suffix rule and the bad character rule do after finding an occurrence of a pattern with period k?

- The good suffix rule would shift P by k positions to the right and then compare all the |P| characters starting from the end of P
- The bad character rule does not apply to matches
- Using the Galil rule we spare |P|-k comparisons

Intuitively, this is enough to prove worst-case linear time because a pattern can have many occurrences in T only if they largely overlap between them; and this happens only if the pattern has a small period.

Comparison between Knuth-Morris-Pratt and Boyer-Moore-Galil

- Both use a sliding window of the same length as the pattern. The window delimits a factor of the text to be examined, and slides along the text from left to right. Not all existing pattern matching algorithms use this framework.

KMP

- $\Theta(|P|+|T|)$ worst-case running time
- P is scanned from left to right

BMG

- $O(|P|+|T|)$ worst-case running time; sublinear expected time
- P is scanned from right to left