Exact Pattern Matching on Strings

Chapter 32 of Cormen’s book, excluding 32.2 and 32.3

Giulia Bernardini
giulia.bernardini@units.it

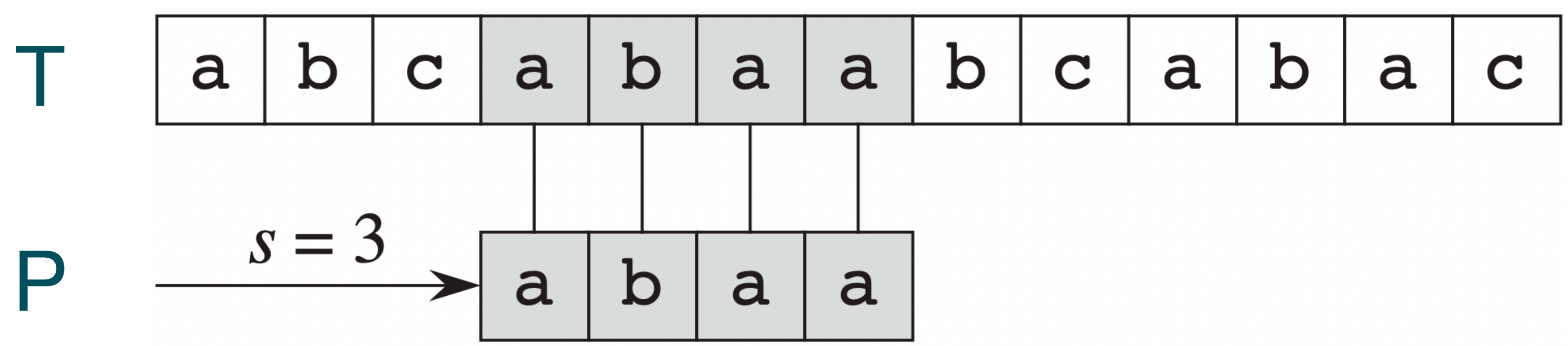
Algorithmic Design
a.y. 2022/2023

Pattern Occurrences

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

The string-matching problem

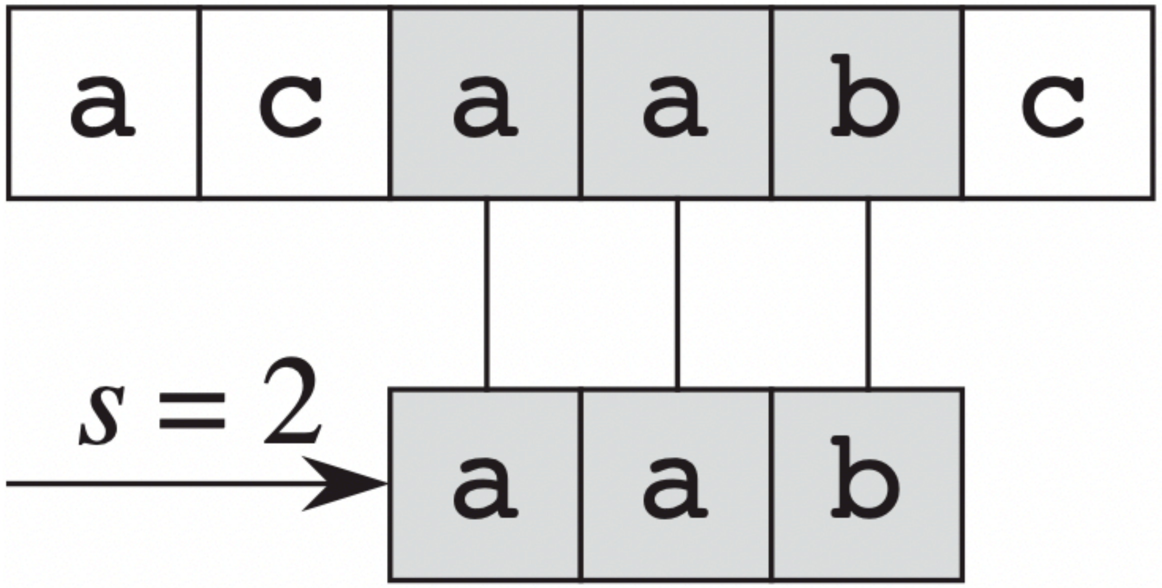
Input: a text T of length n and a pattern P of length m≤n

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



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;
         if i>|P|
            sol.append(s);
   return sol;
```

O(|P|) (while loop)

O(|T|) (for loop)

KMP: 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;

KMP: Preprocessing the pattern

```
COMPUTE_PREFIX(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;
```

- 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

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.

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

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;
```

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

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;
```

If the while loop cannot find a k ∈ $\pi^*[q-1]$ 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 π[q] to 1; otherwise we should leave k alone and set π[q] to 0.

Lines 7–9 set k and π[q] correctly in either case.

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