数据结构与算法（Python）-03+/KMP

刘云淮 Yunhuai.liu@pku.edu.cn

http://www.yunhuai.net/DSA2023/CoursePage/DSA2023.html

北京大学计算机学院

Some slides adapted from https://ics.uci.edu/~goodrich/teach/cs262P/ppt/05-ExactMatch.pptx

Review: Strings

- A **string** is a sequence of characters (indexed from 0)*
- Examples of strings:
    - Python program
    - HTML document
    - DNA sequence
    - Digitized image
- An alphabet S is the set of possible characters for a family of strings
- Example of alphabets:
    - ASCII or Unicode
    - {0, 1}
    - {A, C, G, T}

- Let P be a string of size m
    - A **substring** $P[i : j]$ of P is the subsequence of P consisting of the characters with ranks between i and j
    - A **prefix** of P is a substring of the type $P[0 : i]$
    - A **suffix** of P is a substring of the type $P[i : m - 1]$
- Given strings T (text) and P (pattern), the pattern matching problem consists of finding a substring of T equal to P
- Applications:

Text editors

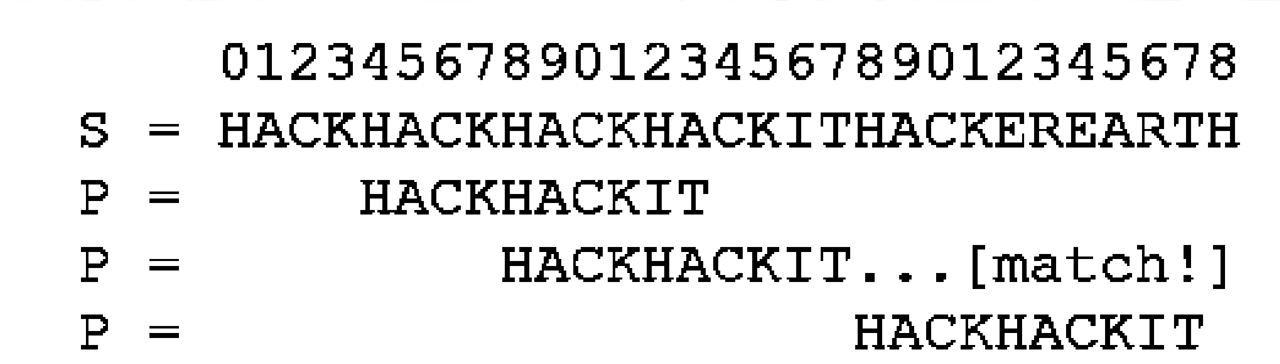
Search engines

Biological research

***Some people index starting from 1.**

Application: fgrep

- Recall that **fgrep** looks for an exact match of a text string in a file.
- So we are interested in fast algorithms for the **exact match** problem: Given a text string, T, of length n, and a pattern string, P, of length m, over an alphabet of size k, find the first (or all) places where a substring of T matches P.



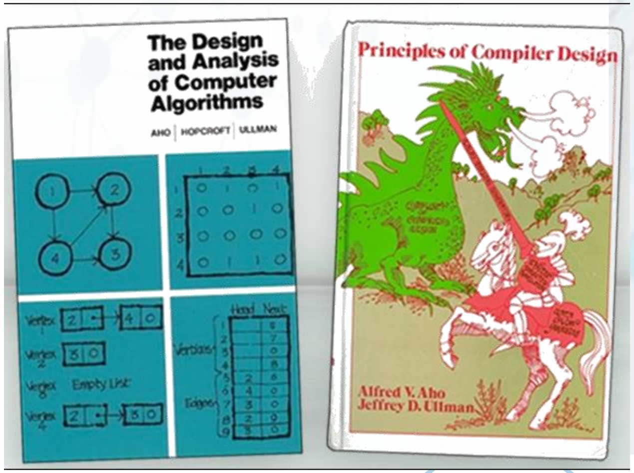
```
    01234567890123456789012345678
S = HACKHACKHACKHACKITHACKEREARTH
P =     HACKHACKIT
P =         HACKHACKIT...[match!]
P =                   HACKHACKIT
```

Image from https://www.hackerearth.com/practice/notes/exact-string-matching-algorithms/

Alfred Aho

- 1975: Invented fgrep
- …*
- 2020: received the Turing Award

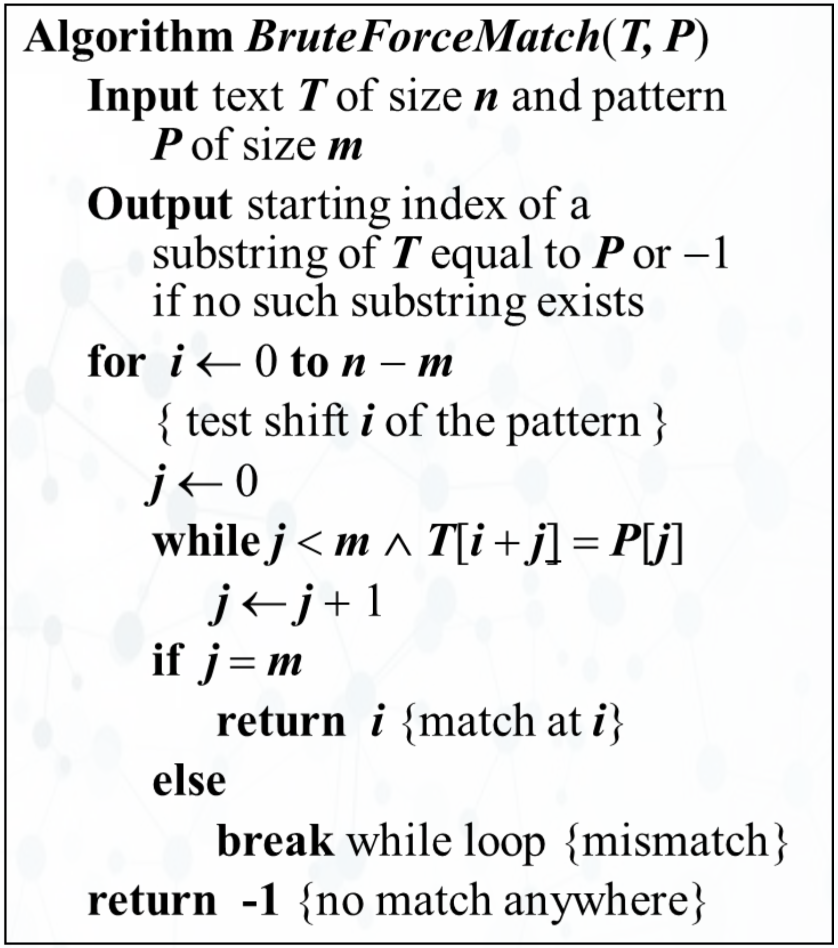
* Also invented text processing techniques used in every modern source-code compiler and co-authored two influential textbooks.



Images from https://awards.acm.org/about/2020-turing

Brute-force Pattern Matching

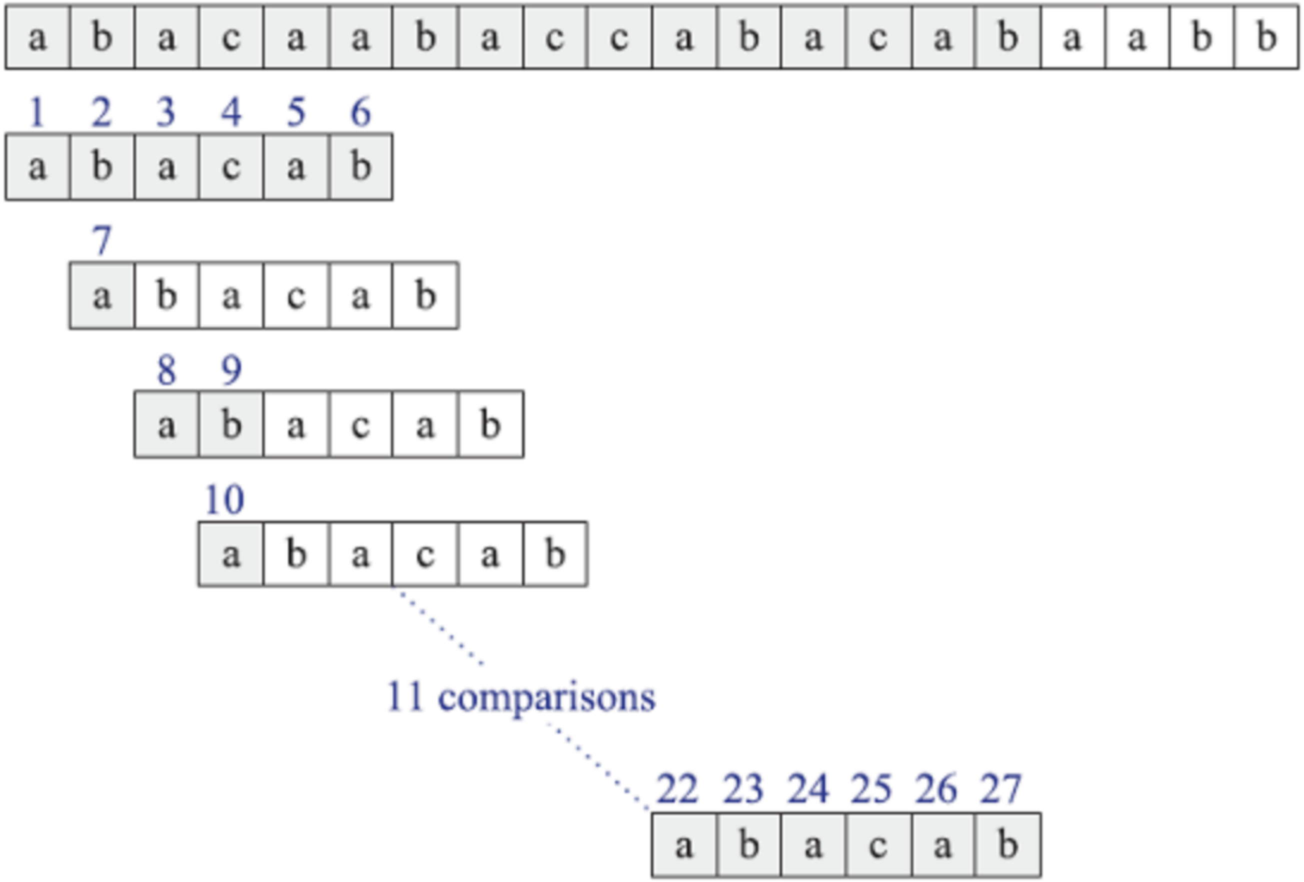
- The Brute-force (Naïve) pattern matching algorithm compares the pattern P with the text T for each possible shift of P relative to T, until either
  - a match is found, or
  - all placements of the pattern have been tried
- Brute-force pattern matching runs in time $O(nm)$
- Example of worst case:
  - T = aaa … ah
  - P = aaah
  - may occur in images and DNA sequences



```
Algorithm BruteForceMatch(T, P)
    Input text T of size n and pattern
        P of size m
    Output starting index of a
        substring of T equal to P or −1
        if no such substring exists
    for  i ← 0 to n − m
        { test shift i of the pattern }
        j ← 0
        while j < m ∧ T[i + j] = P[j]
            j ← j + 1
        if  j = m
            return  i {match at i}
        else
            break while loop {mismatch}
    return  -1 {no match anywhere}
```

Brute-Force Matching Example

- Trying every possible position for a match:



Expected-case Analysis for Brute-force

- The worst-case running time for Brute-force algorithm O(mn), but it runs in expected linear time for random strings.
- Suppose P and T are strings of m and n characters respectively chosen uniformly and independently at random from an alphabet of size k.
- Let $X_{i,j}$ be a random variable that is 1 if and only if P[i] is compared to T[j], and note that probability $X_{i,j}$ is 1 is $1/k^i$ because this occurs when we have i character matches.
- By the linearity of expectation, the expected number of comparisons for any T[j] is therefore

$$1/k + 1/k^2 + 1/k^3 + \cdots + 1/k^m,$$

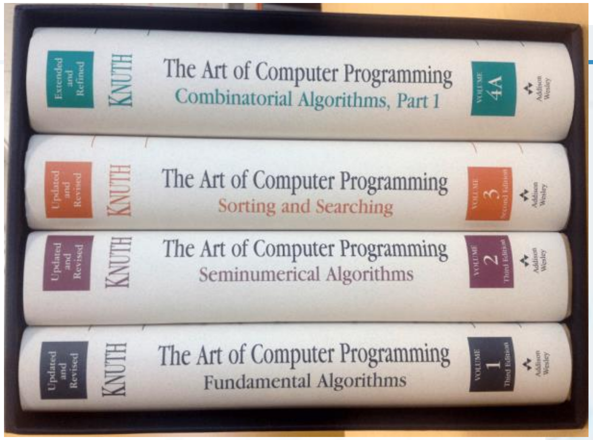
which is at most 2.

- Thus, the expected number of comparisons is at most 2n.

Expected-case Analysis for Exact String Matching is Problematic

- If a pattern string P and text string T are strings of characters chosen uniformly and independently at random from an alphabet of size k, then the probability that P appears anywhere in T is at most n/k^m.
- For example, if n=1000, m=10, and k=50, then the probability of a match of P in T is about 1 in 10 trillion!
- In this case, a fast (and very accurate) exact matching algorithm is:

Donald Knuth

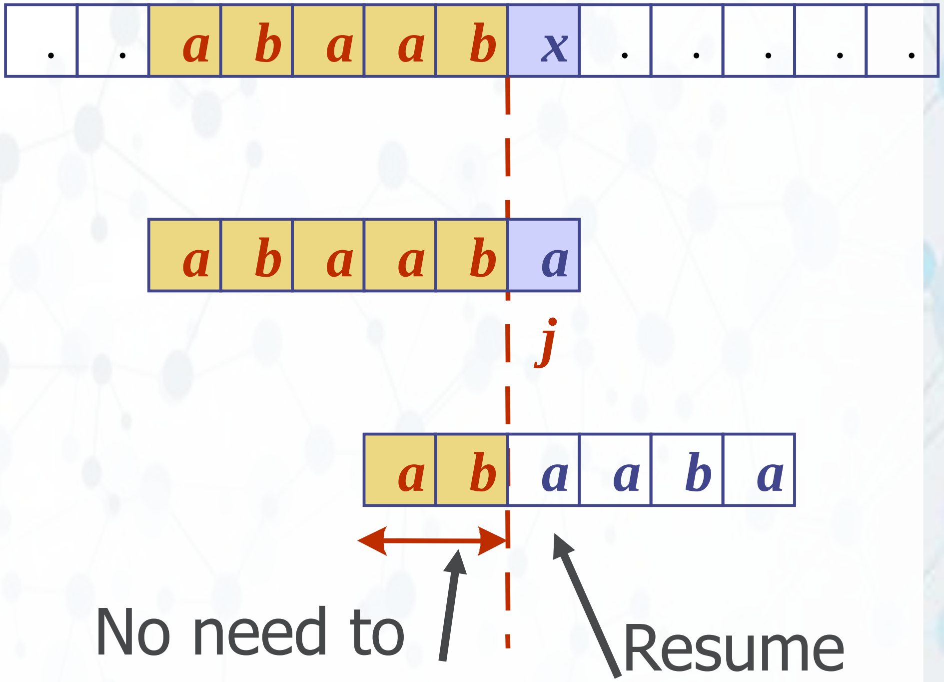


- 1973: Discovered the KMP algorithm (which was also published in a technical report by Morris and Pratt in 1970—all three published a joint paper describing the algorithm in 1977).
- 1974: Received the Turing Award.
- He is also known for his book series, “The Art of Computer Programming,” which formalized and popularized algorithm analysis (e.g., the “big O”).

Image from https://en.wikipedia.org/wiki/Donald_Knuth

The KMP Algorithm

- Consider the comparison of a pattern with a text as in the brute-force algorithm.
- When a mismatch occurs, what is the **most** we can shift the pattern so as to avoid redundant comparisons?
- Answer: the **largest** prefix of $P[0..j]$ that is a suffix of $P[1..j]$
- This approach is similar to the NFA-to-DFA approach, but is implemented more efficiently.



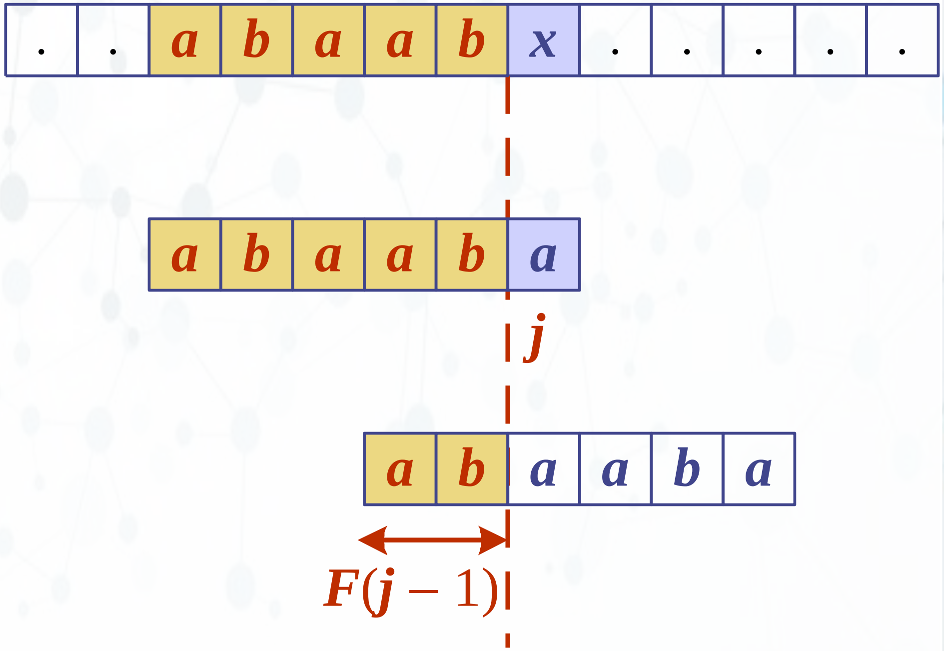
No need to repeat these comparisons

Resume comparing here

The KMP Failure Function

- Knuth-Morris-Pratt’s algorithm preprocesses the pattern to find matches of prefixes of the pattern with the pattern itself
- The **failure function** $F(j)$ is defined as the length of the longest prefix of $P[0..j]$ that is also a suffix of $P[1..j]$
- Knuth-Morris-Pratt’s algorithm modifies the brute-force algorithm so that if a mismatch occurs at $P[j] \neq T[i]$ and $j > 0$, we set $j \leftarrow F(j-1)$

j	0	1	2	3	4	5
***P*[*j*]**	***a***	***b***	***a***	***a***	***b***	***a***
***F*(*j*)**	0	0	1	1	2	3

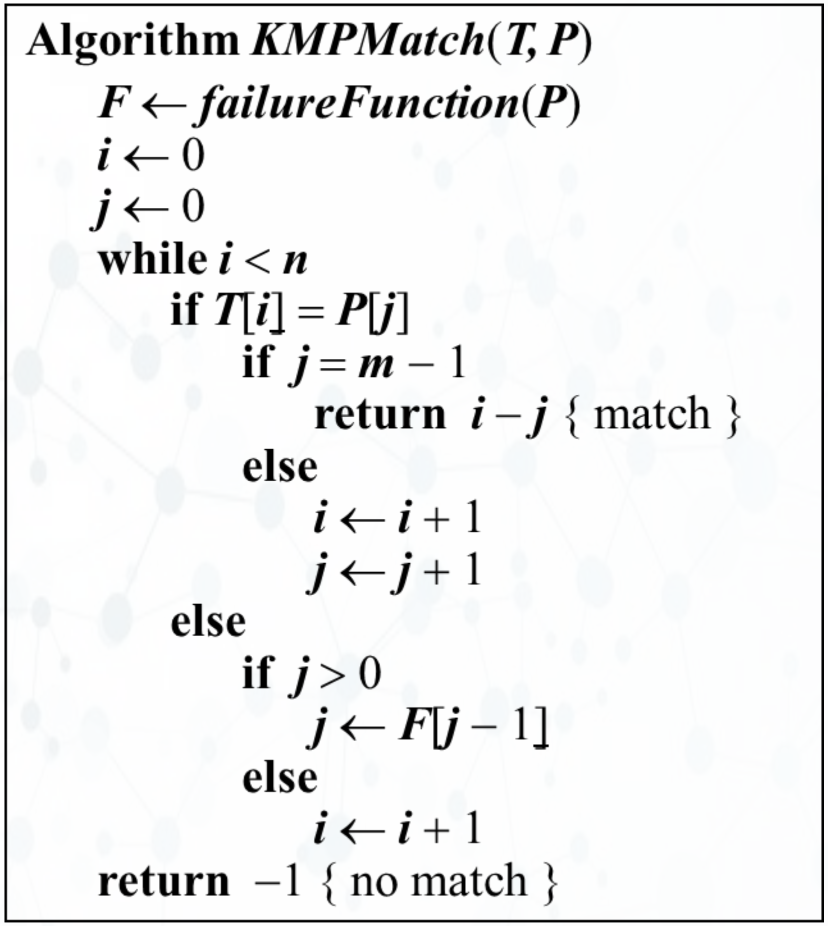


The KMP Algorithm

- The failure function can be represented by an array and can be computed in $O(m)$ time
- At each iteration of the while-loop, either

  i increases by one, or

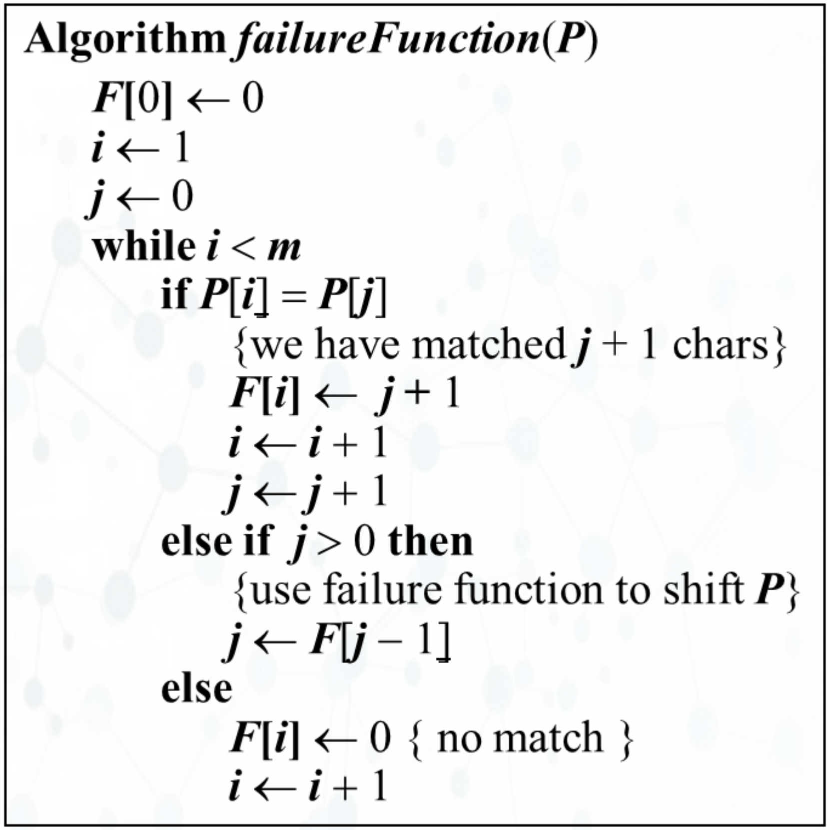
  the shift amount $i - j$ increases by at least one (observe that $F(j - 1) < j$)
- Hence, there are no more than $2n$ iterations of the while-loop
- Thus, KMP’s algorithm runs in optimal time $O(m + n)$



```
Algorithm KMPMatch(T, P)
    $F \leftarrow failureFunction(P)$
    $i \leftarrow 0$
    $j \leftarrow 0$
    while $i < n$
        if $T[i] = P[j]$
            if $j = m - 1$
                return $i - j$ { match }
            else
                $i \leftarrow i + 1$
                $j \leftarrow j + 1$
        else
            if $j > 0$
                $j \leftarrow F[j - 1]$
            else
                $i \leftarrow i + 1$
    return $-1$ { no match }
```

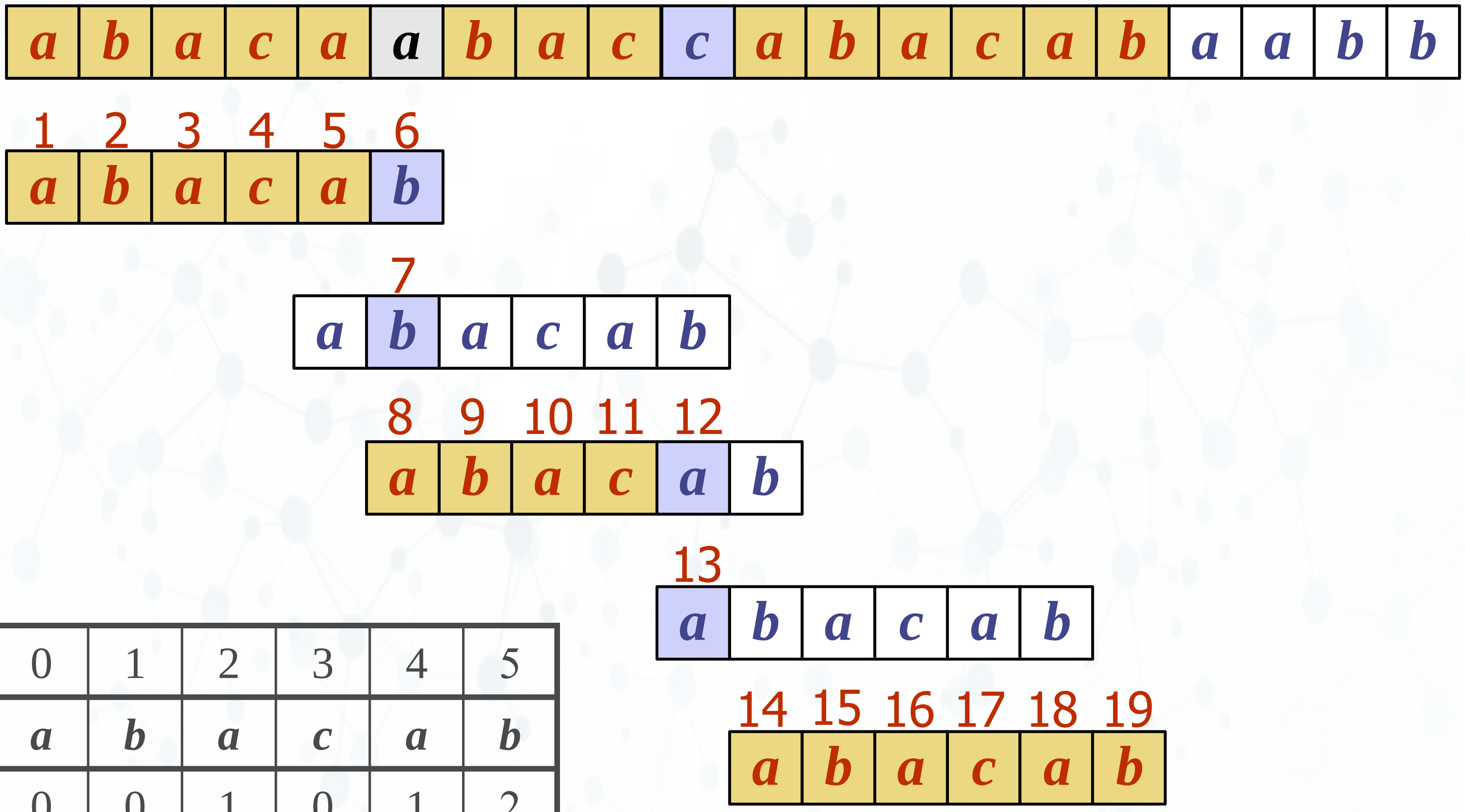
Computing the Failure Function

- The failure function can be represented by an array and can be computed in $O(m)$ time
- The construction is similar to the KMP algorithm itself
- At each iteration of the while-loop, either
    - i increases by one, or
    - the shift amount $i - j$ increases by at least one (observe that $F(j - 1) < j$)
- Hence, there are no more than $2m$ iterations of the while-loop



```
Algorithm failureFunction(P)
    $F[0] \leftarrow 0$
    $i \leftarrow 1$
    $j \leftarrow 0$
    while $i < m$
        if $P[i] = P[j]$
            {we have matched $j + 1$ chars}
            $F[i] \leftarrow j + 1$
            $i \leftarrow i + 1$
            $j \leftarrow j + 1$
        else if $j > 0$ then
            {use failure function to shift $P$}
            $j \leftarrow F[j - 1]$
        else
            $F[i] \leftarrow 0$ { no match }
            $i \leftarrow i + 1$
```

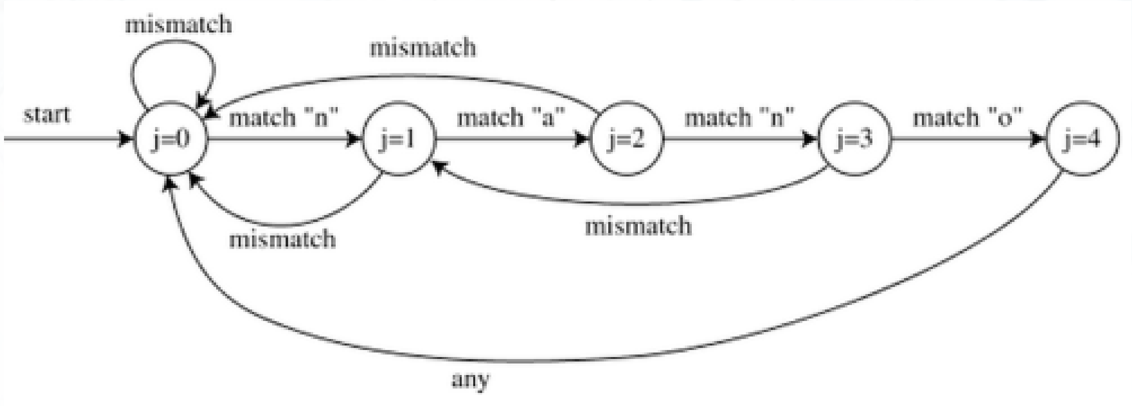
Example



j	0	1	2	3	4	5
P[*j*]	*a*	*b*	*a*	*c*	*a*	*b*
F(*j*)	0	0	1	0	1	2

Summary for KMP

- Thus, the KMP algorithm runs in O(m+n) time for constant-size alphabets.
- The KMP algorithm mimics the NFA-to-DFA algorithm but doesn’t have extra preprocessing time because it builds a failure table for the **longest** prefix that matches a suffix at each pattern position, rather than encoding all prefixes.



State machine of “nano”

Image from http://www.zrzahid.com/linear-time-string-matching-using-kmp-matching-algorithm/