SE274 Data Structure

Lecture 10: Text Processing

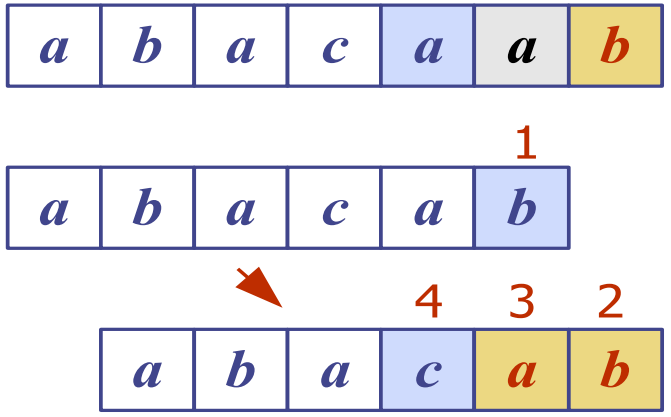
(textbook: Chapter 13)

May *27*, 2020

Instructor: Sunjun Kim

Information&Communication Engineering, DGIST

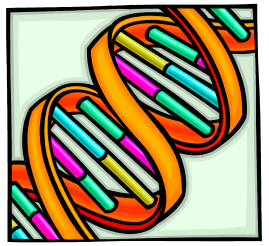
Pattern Matching



Strings

- A string is a sequence of characters
- Examples of strings:
  - Python program
  - HTML document
  - DNA sequence
  - Digitized image
- An alphabet Σ is the set of possible characters for a family of strings
- Example of alphabets:
  - ASCII
  - 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

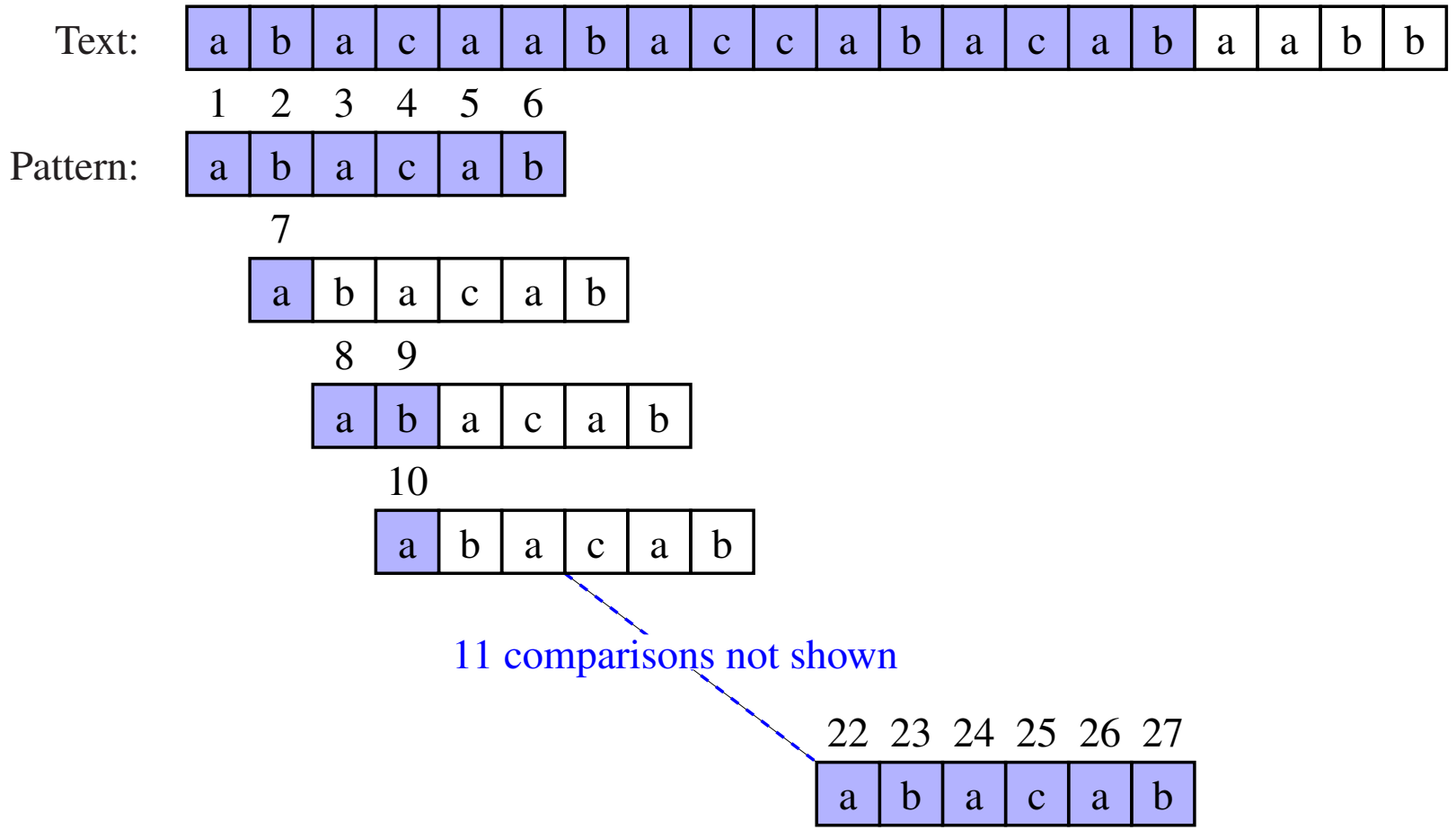
Brute-Force Pattern Matching



- The brute-force 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:
  - $T = aaa \ldots ah$
  - $P = aaah$
  - may occur in images and DNA sequences
  - unlikely in English text

```
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 Pattern Matching



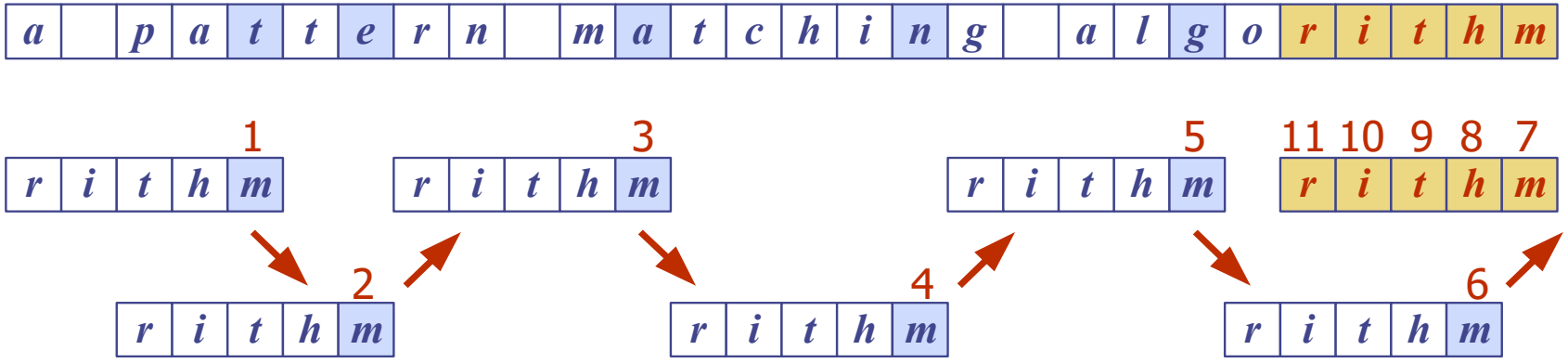
Boyer-Moore Heuristics

- The Boyer-Moore's pattern matching algorithm is based on two heuristics

  Looking-glass heuristic: Compare P with a subsequence of T moving backwards

  Character-jump heuristic: When a mismatch occurs at $T[i] = c$
  - If P contains c, shift P to align the last occurrence of c in P with $T[i]$
  - Else, shift P to align $P[0]$ with $T[i+1]$

- Example



Last-Occurrence Function

- Boyer-Moore's algorithm preprocesses the pattern P and the alphabet Σ to build the last-occurrence function L mapping Σ to integers, where $L(c)$ is defined as
  - the largest index i such that $P[i] = c$ or
  - -1 if no such index exists
- Example:
  - $\Sigma = \{a, b, c, d\}$
  - $P = abacab$

c	a	b	c	d
$L(c)$	4	5	3	-1

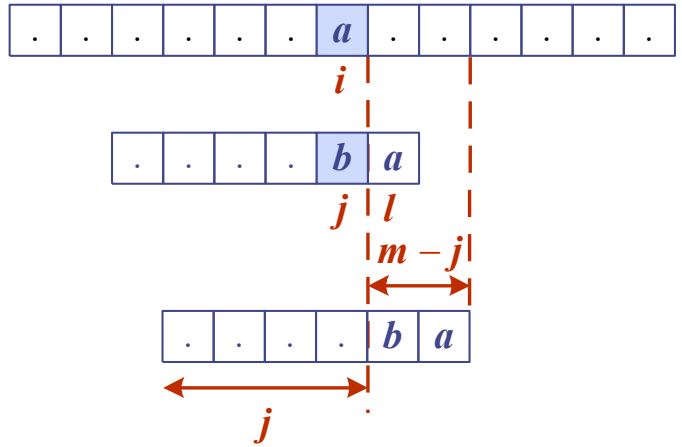
- The last-occurrence function can be represented by an array indexed by the numeric codes of the characters
- The last-occurrence function can be computed in time $O(m + s)$, where m is the size of P and s is the size of Σ

The Boyer-Moore Algorithm

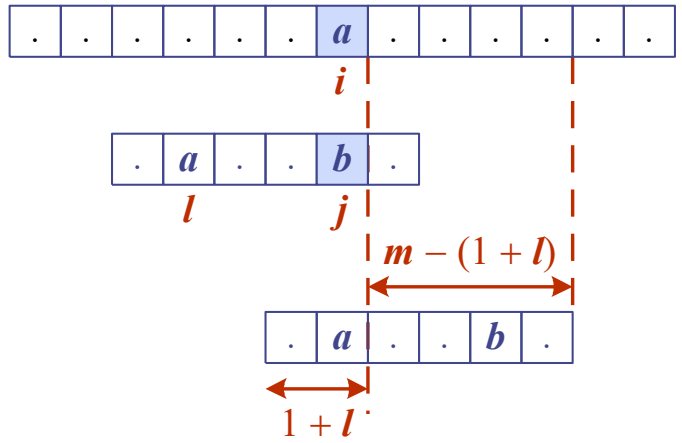
i: current index in T
j: current index in P
l: $L[T[i]]$

```
Algorithm BoyerMooreMatch(T, P, Σ)
    L ← lastOccurenceFunction(P, Σ)
    i ← m − 1
    j ← m − 1
    repeat
        if T[i] = P[j]
            if j = 0
                return i { match at i }
            else
                i ← i − 1
                j ← j − 1
        else
            { character-jump }
            l ← L[T[i]]
            i ← i + m − min(j, 1 + l)
            j ← m − 1
    until i > n − 1
    return −1 { no match }
```

Case 1: $j \leq 1 + l$

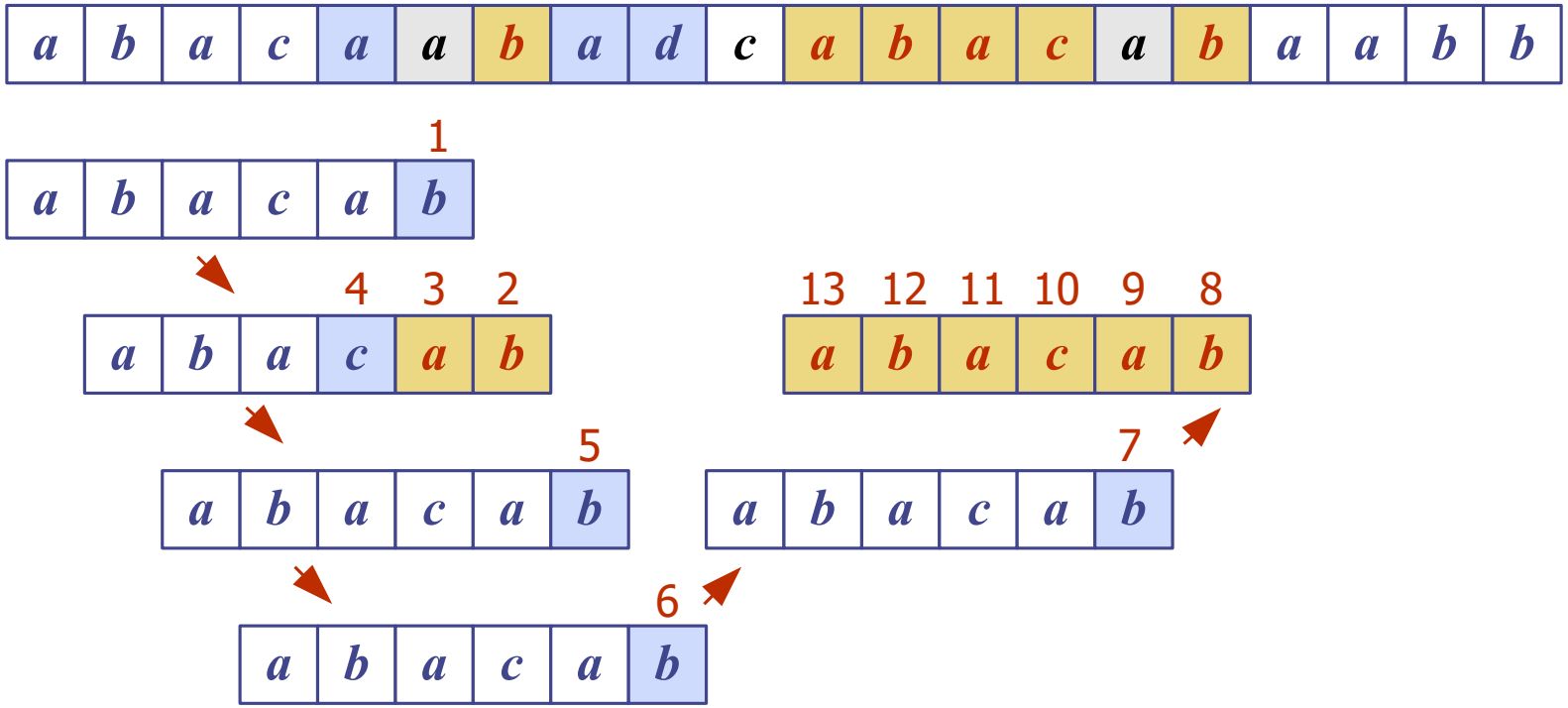


Case 2: $1 + l \leq j$



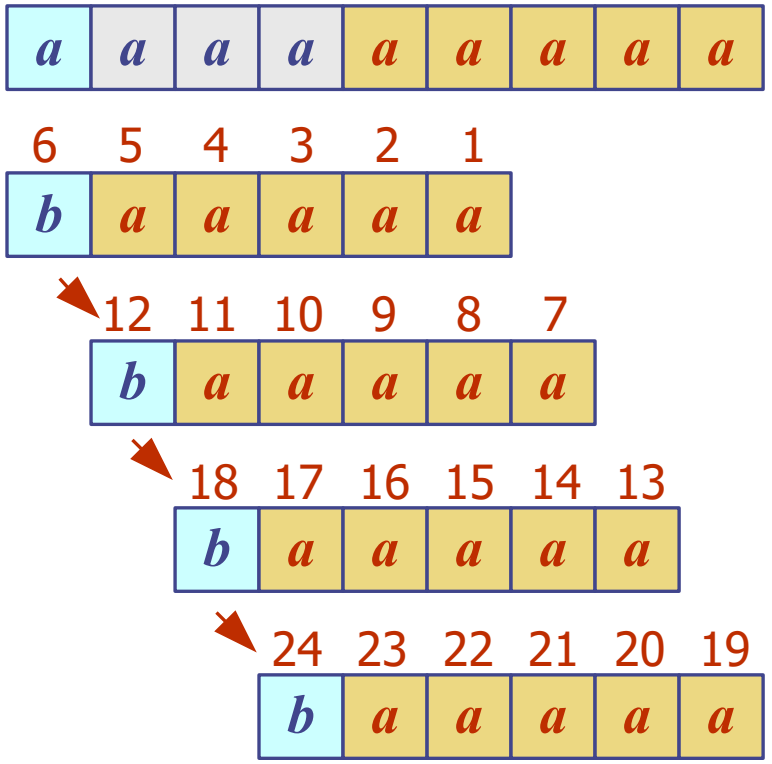
Example

c	a	b	c	d
$L(c)$	4	5	3	−1



Analysis

- Boyer-Moore's algorithm runs in time $O(nm + s)$
- Example of worst case:
  - $T = aaa \ldots a$
  - $P = baaa$
- The worst case may occur in images and DNA sequences but is unlikely in English text
- Boyer-Moore's algorithm is significantly faster than the brute-force algorithm on English text

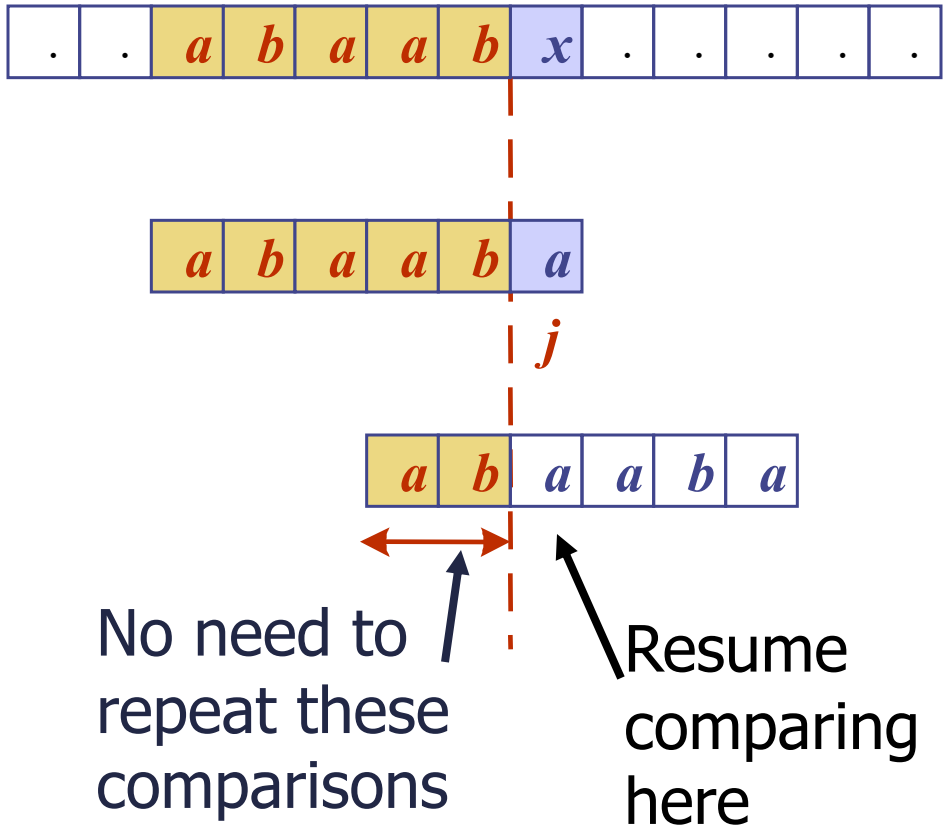


Python Implementation

```
1  def find_boyer_moore(T, P):
2    """Return the lowest index of T at which substring P begins (or else -1)."""
3    n, m = len(T), len(P)                    # introduce convenient notations
4    if m == 0: return 0                      # trivial search for empty string
5    last = { }                               # build 'last' dictionary
6    for k in range(m):
7      last[ P[k] ] = k                       # later occurrence overwrites
8    # align end of pattern at index m-1 of text
9    i = m−1                                  # an index into T
10   k = m−1                                  # an index into P
11   while i < n:
12     if T[i] == P[k]:                       # a matching character
13       if k == 0:
14         return i                           # pattern begins at index i of text
15       else:
16         i −= 1                             # examine previous character
17         k −= 1                             # of both T and P
18     else:
19       j = last.get(T[i], −1)               # last(T[i]) is -1 if not found
20       i += m − min(k, j + 1)               # case analysis for jump step
21       k = m − 1                            # restart at end of pattern
22   return −1
```

The KMP Algorithm

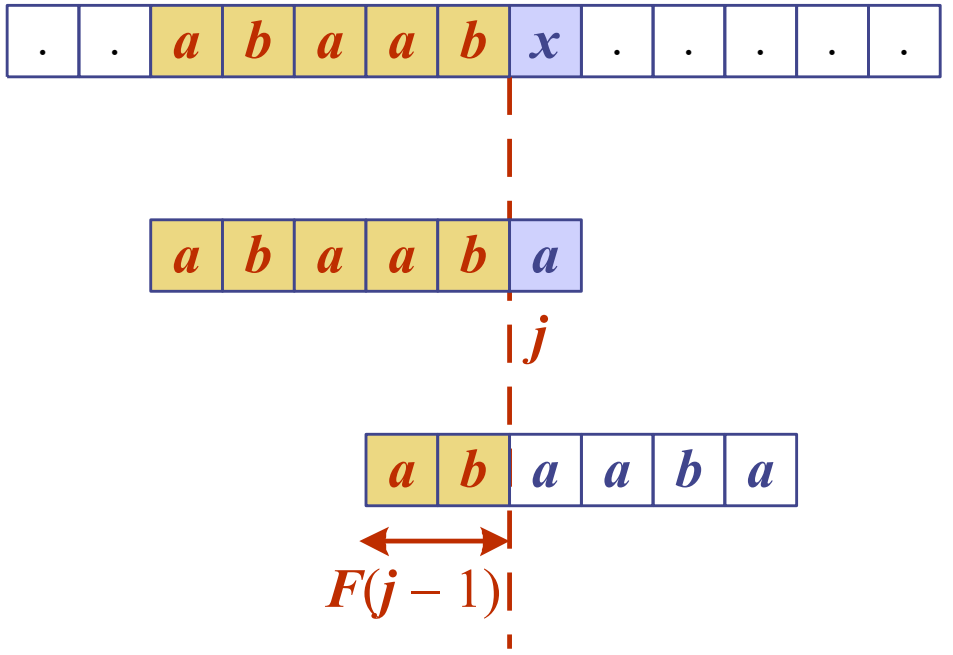
- Knuth-Morris-Pratt's algorithm compares the pattern to the text in left-to-right, but shifts the pattern more intelligently than the brute-force algorithm.
- When a mismatch occurs, what is the **most** we can shift the pattern so as to avoid redundant comparisons?
- Answer: the largest prefix of $P[0..j]$ that is a suffix of $P[1..j]$



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 size of the largest 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 $P[j] \neq T[i]$ 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

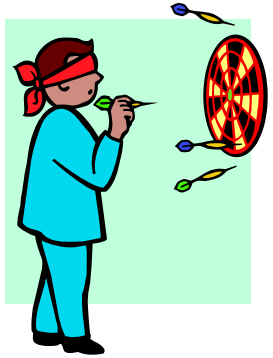


The KMP Algorithm

- 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 ← failureFunction(P)
    i ← 0
    j ← 0
    while i < n
        if T[i] = P[j]
            if j = m − 1
                return i − j { match }
            else
                i ← i + 1
                j ← j + 1
        else
            if j > 0
                j ← F[j − 1]
            else
                i ← i + 1
    return −1 { no match }
```

Computing the Failure Function

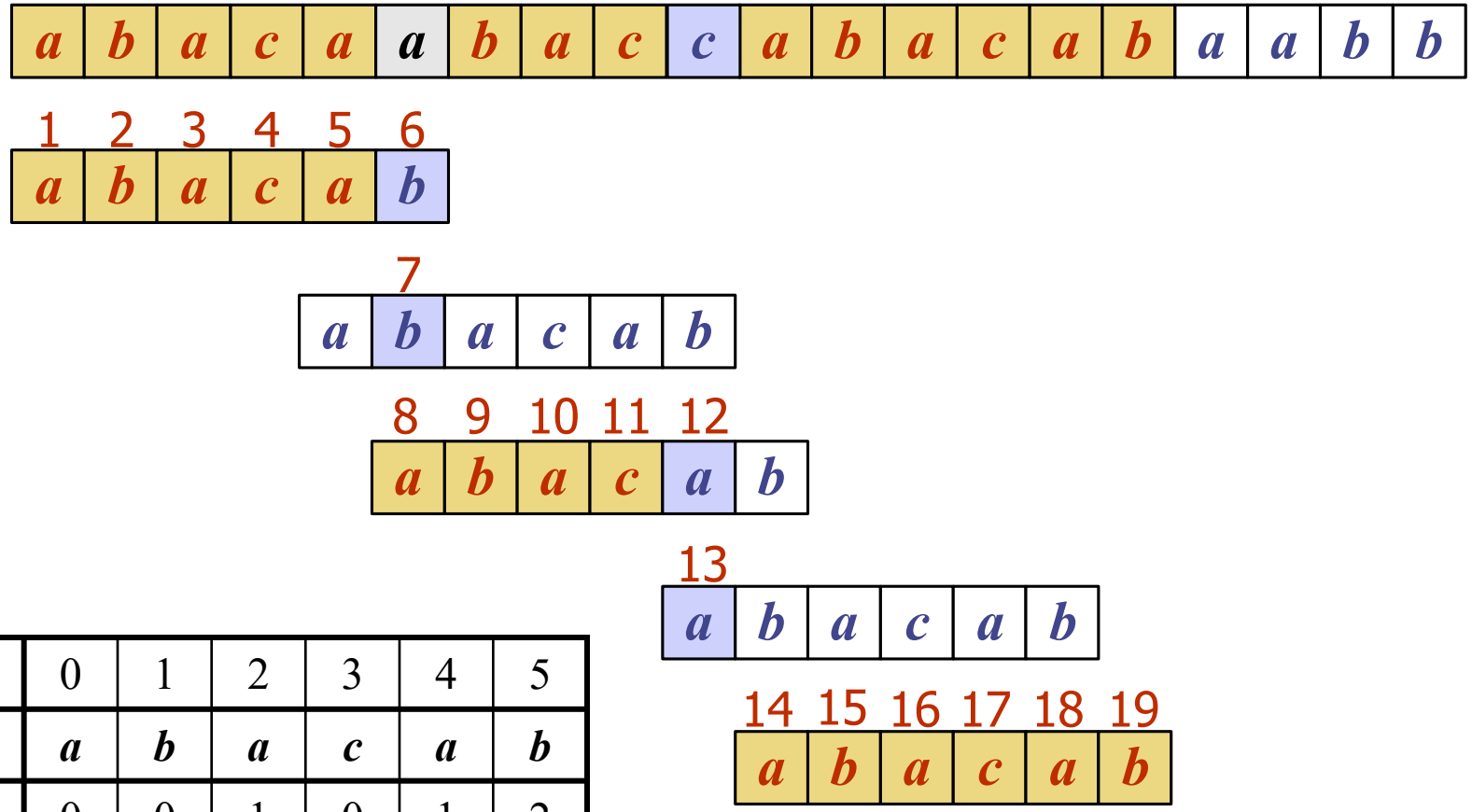


- The failure function can be represented by an array and can be computed in $O(m)$ time
- The construction is similar to the KMP algorithm itself
- At each iteration of the while-loop, either
  - i increases by one, or
  - the shift amount $i - j$ increases by at least one (observe that $F(j-1) < j$)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

Algorithm *failureFunction*(P)

```
F[0] ← 0
i ← 1
j ← 0
while i < m
    if P[i] = P[j]
        {we have matched j + 1 chars}
        F[i] ← j + 1
        i ← i + 1
        j ← j + 1
    else if j > 0 then
        {use failure function to shift P}
        j ← F[j − 1]
    else
        F[i] ← 0 { no match }
        i ← 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

Python Implementation

```
1  def find_kmp(T, P):
2    """Return the lowest index of T at which substring P begins (or else -1)."""
3    n, m = len(T), len(P)                    # introduce convenient notations
4    if m == 0: return 0                      # trivial search for empty string
5    fail = compute_kmp_fail(P)               # rely on utility to precompute
6    j = 0                                    # index into text
7    k = 0                                    # index into pattern
8    while j < n:
9      if T[j] == P[k]:                       # P[0:1+k] matched thus far
10       if k == m - 1:                       # match is complete
11         return j - m + 1
12     j += 1                                 # try to extend match
13     k += 1
14   elif k > 0:
15     k = fail[k-1]                          # reuse suffix of P[0:k]
16   else:
17     j += 1
18   return -1                                # reached end without match
```

```
1  def compute_kmp_fail(P):
2    """Utility that computes and returns KMP 'fail' list."""
3    m = len(P)
4    fail = [0] * m                           # by default, presume overlap of 0 everywhere
5    j = 1
6    k = 0
7    while j < m:                             # compute f(j) during this pass, if nonzero
8      if P[j] == P[k]:                       # k + 1 characters match thus far
9        fail[j] = k + 1
10       j += 1
11       k += 1
12     elif k > 0:                            # k follows a matching prefix
13       k = fail[k-1]
14     else:                                  # no match found starting at j
15       j += 1
16   return fail
```

Further readings on textbook:

- Longest Common Subsequence (LCS) problem
- Dynamic programming
- Greedy Algorithm

➔ You will learn them in detail on Algorithm course

Regular Expression (Regex)

Text: `purple alice-b@google.com monkey dishwasher`

Regex: `[\w\.-]+@[\w\.-]+`

Match: `alice-b@google.com`

Content Ref: https://developers.google.com/edu/python/regular-expressions

Regular Expression

- Shortend as *regex*, or *regexp*
- Regular expressions are a powerful language for matching **text patterns**.

- Pattern example:
  - IP address: ###.###.###.###, where the numbers should be 0-255
  - E-mail address: [name]@[domain],
    - where [name] should only contains alphabet, number, dot, and hyphen
    - where [domain] should only contains alphabet, number, hyphen, and at least one dot.
  - HTML Tag: <tag attribute=value>container</tag>

Basic patterns

Pattern	Description
a, X, 9, …	ordinary characters just match themselves exactly, except meta-characters: **. ^ $ * + ? { \ | ( )** (details later)
. (a period)	matches any single character except newline '\n'
\w, \W	\w: matches a "word" character: a single letter or digit or underbar [a-zA-Z0-9_]. \W: matches any non-word character.
\b	Boundary between word and non-word
\s, \S	\s: matches a single whitespace character (space, newline, return, tab, form [\n\r\t\f]. \S: matches any non-whitespace character.
\t, \n, \r	tab, newline, return
\d	A decimal digit [0-9]
^, $	^ = start, $ = end -- match the start or end of the string
\ (escape)	\ -- inhibit the "specialness" of a character. So, for example, use \. to match a period or \\ to match a slash. If you are unsure if a character has special meaning, such as '@', you can put a slash in front of it, \@, to make sure it is treated just as a character.

Python Regex

- ‘re’ package
- https://docs.python.org/3/library/re.html

```
import re
…
```

```
str = 'an example word:cat!!'
match = re.search(r'word:\w\w\w', str)
# If-statement after search() tests if it succeeded
if match:
  print 'found', match.group() ## 'found word:cat'
else:
  print 'did not find'
```

Basic Examples

```
## Search for pattern 'iii' in string 'piiig'.
## All of the pattern must match, but it may appear anywhere.
## On success, match.group() is matched text.
match = re.search(r'iii', 'piiig') # found, match.group() == "iii"
match = re.search(r'igs', 'piiig') # not found, match == None

## . = any char but \n
match = re.search(r'..g', 'piiig') # found, match.group() == "iig"

## \d = digit char, \w = word char
match = re.search(r'\d\d\d', 'p123g') # found, match.group() == "123"
match = re.search(r'\w\w\w', '@@abcd!!') # found, match.group() == "abc"
```

Quantifier

Pattern	Description
+	1 or more occurrences of the pattern to its left, e.g. 'i+' = one or more i's
*	0 or more occurrences of the pattern to its left
?	match 0 or 1 occurrences of the pattern to its left
{num}	Numeric quantifier {3}: exactly 3 repeats {3,}: 3 or more repeats {3,6}: between 3 and 6 repeats

Basic Examples (cont.)

```
## i+ = one or more i's, as many as possible.
match = re.search(r'pi+', 'piiig') # found, match.group() == "piii"

## Finds the first/leftmost solution, and within it drives the +
## as far as possible (aka 'leftmost and largest').
## In this example, note that it does not get to the second set of i's.
match = re.search(r'i+', 'piigiiii') # found, match.group() == "ii"

## \s* = zero or more whitespace chars
## Here look for 3 digits, possibly separated by whitespace.
match = re.search(r'\d\s*\d\s*\d', 'xx1 2   3xx') # found, match.group() == "1 2   3"
match = re.search(r'\d\s*\d\s*\d', 'xx12  3xx') # found, match.group() == "12  3"
match = re.search(r'\d\s*\d\s*\d', 'xx123xx') # found, match.group() == "123"

## ^ = matches the start of string, so this fails:
match = re.search(r'^b\w+', 'foobar') # not found, match == None
## but without the ^ it succeeds:
match = re.search(r'b\w+', 'foobar') # found, match.group() == "bar"
```

Square Braket []: set

- Square brackets can be used to indicate a set of chars, so [abc] matches 'a' or 'b' or 'c'. The codes \w, \s etc. work inside square brackets too with the one exception that dot (.) just means a literal dot.

```
match = re.search(r'[\w.-]+@[\w.-]+', str)
if match:
  print match.group()  ## 'alice-b@google.com'
```

Parenthesis (): group extraction

- The "group" feature of a regular expression allows you to pick out parts of the matching text. Suppose for the emails problem that we want to extract the username and host separately. To do this, add parenthesis ( ) around the username and host in the pattern, like this: r'([\w.-]+)@([\w.-]+)'. In this case, the parenthesis do not change what the pattern will match, instead they establish logical "groups" inside of the match text.

```
str = 'purple alice-b@google.com monkey dishwasher'
match = re.search(r'([\w.-]+)@([\w.-]+)', str)
if match:
  print match.group()   ## 'alice-b@google.com' (the whole match)
  print match.group(1)  ## 'alice-b' (the username, group 1)
  print match.group(2)  ## 'google.com' (the host, group 2)
```

re.findall

re.**findall**(*pattern*, *string*, *flags=0*)

Return all non-overlapping matches of *pattern* in *string*, as a list of strings. The *string* is scanned left-to-right, and matches are returned in the order found. If one or more groups are present in the pattern, return a list of groups; this will be a list of tuples if the pattern has more than one group. Empty matches are included in the result.

```
## Suppose we have a text with many email addresses
str = 'purple alice@google.com, blah monkey bob@abc.com blah dishwasher'

## Here re.findall() returns a list of all the found email strings
emails = re.findall(r'[\w\.-]+@[\w\.-]+', str) ## ['alice@google.com', 'bob@abc.com']
for email in emails:
  # do something with each found email string
  print email
```

Substitution

`re.` **sub**(*pattern*, *repl*, *string*, *count=0*, *flags=0*)

Return the string obtained by replacing the leftmost non-overlapping occurrences of *pattern* in *string* by the replacement *repl*. If the pattern isn't found, *string* is returned unchanged. *repl* can be a string or a function; if it is a string, any backslash escapes in it are processed. That is, `\n` is converted to a single newline character, `\r` is converted to a carriage return, and so forth. Unknown escapes of ASCII letters are reserved for future use and treated as errors. Other unknown escapes such as `\&` are left alone. Backreferences, such as `\6`, are replaced with the substring matched by group 6 in the pattern. For example:

```
str = 'purple alice@google.com, blah monkey bob@abc.com blah dishwasher'
## re.sub(pat, replacement, str) -- returns new string with all replacements,
## \1 is group(1), \2 group(2) in the replacement
print re.sub(r'([\w\.-]+)@([\w\.-]+)', r'\1@yo-yo-dyne.com', str)
## purple alice@yo-yo-dyne.com, blah monkey bob@yo-yo-dyne.com blah dishwasher
```

Regex Study material

- Baby name exercise: https://developers.google.com/edu/python/exercises/baby-names
- University of Trieste, Machine Learning Lab RegexPlay: http://play.inginf.units.it/
- Online Regex matcher & parser: https://regex101.com/

http://play.inginf.units.it/

Your regex

(?<=\()([\da-f]{2}:)+[\da-f]{2}(?<!\))

Submit

2/12 Task

03:57 Time

100.0 F-measure

Jan 13 00:48:59: DROP service 68->67(udp) from 213.92.153.167 to 69.43.107.219, prefix: "spoof iana-0/8" (in: eth0 69.43.112.233(38:f8:b7:90:45:92):68 -> 217.70.100.113(00:21:87:79:9c:d9):67 UDP len:576 ttl:64)

Jan 13 12:02:48: ACCEPT service dns from 74.125.186.208 to firewall(pub-nic-dns), prefix: "none" (in: eth0 74.125.186.208(00:1a:e3:52:5d:8e):36008 -> 140.105.63.158(00:1a:9a:86:2e:62):53 UDP len:82 ttl:38)

Jan 13 17:44:52: DROP service 68->67(udp) from 172.45.240.237 to 217.70.177.60, prefix: "spoof iana-0/8" (in: eth0 216.34.90.16(00:21:91:fe:a2:6f):68 -> 69.43.85.253(00:07:e1:7c:53:db):67 UDP len:328 ttl:64)

Jan 13 17:52:08: ACCEPT service http from 213.121.184.130 to firewall(pub-nic), prefix: "none" (in: eth0 213.121.184.130(00:05:2e:6a:a4:14):8504 -> 140.105.63.164(00:60:11:92:ed:1b):80 TCP flags: ****S* len:52 ttl:109)

Jan 14 04:56:08: DROP service 68->67(udp) from 217.70.196.185 to 217.70.92.217, prefix: "spoof iana-0/8" (in: eth0 69.43.195.65(00:26:f4:fd:77:d1):68 -> 172.45.101.249(bc:b8:52:d0:55:33):67 UDP len:576 ttl:64)

Jan 14 06:03:01: DROP service 68->67(udp) from 217.70.20.228 to 213.92.27.87, prefix: "spoof iana-0/8" (in: eth0 216.34.214.4(0c:71:5d:52:6f:65):68 -> 172.45.70.44(00:00:6b:ed:5e:cf):67 UDP len:328 ttl:64)

Jan 12 17:19:19: DROP service 68->67(udp) from 213.92.192.102 to 216.34.131.104, prefix: "spoof iana-0/8" (in: eth0 213.92.247.248(00:26:0c:9c:60:d1):68 -> 69.43.186.115(00:00:9a:48:ab:b8):67 UDP len:576 ttl:64)

Jan 13 10:03:14: DROP service 68->67(udp) from 213.92.20.178 to 216.34.129.47, prefix: "spoof iana-0/8" (in: eth0 213.92.173.212(00:18:12:c1:5a:a4):68 -> 172.45.188.138(3c:83:b5:65:85:ba):67 UDP len:328 ttl:64)

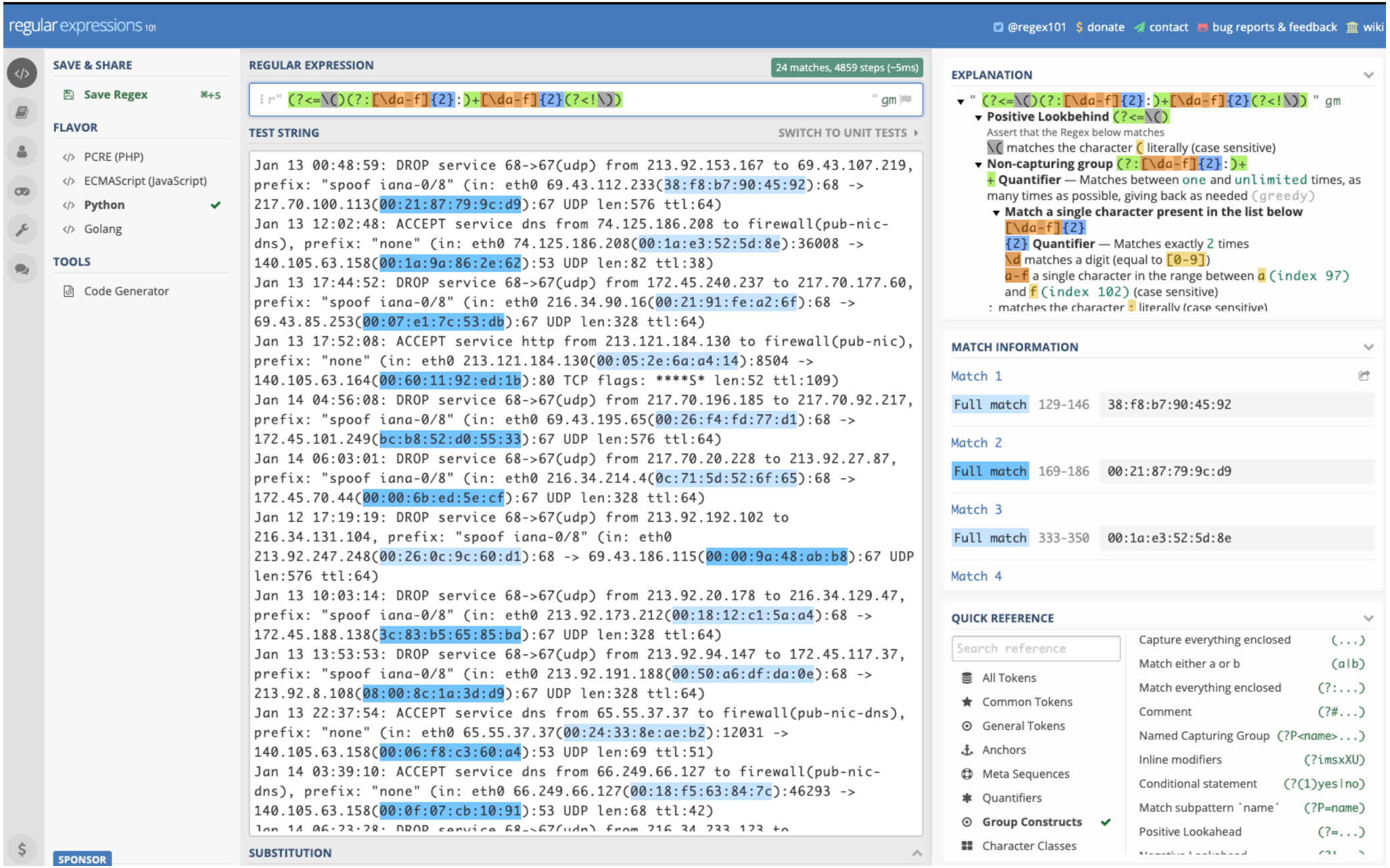
Jan 13 13:53:53: DROP service 68->67(udp) from 213.92.94.147 to 172.45.117.37, prefix: "spoof iana-0/8" (in: eth0 213.92.191.188(00:50:a6:df:da:0e):68 -> 213.92.8.108(08:00:8c:1a:3d:d9):67 UDP len:328 ttl:64)

Jan 13 22:37:54: ACCEPT service dns from 65.55.37.37 to firewall(pub-nic-dns), prefix: "none" (in: eth0 65.55.37.37(00:24:33:8e:ae:b2):12031 -> 140.105.63.158(00:06:f8:c3:60:a4):53 UDP len:69 ttl:51)

Jan 14 03:39:10: ACCEPT service dns from 66.249.66.127 to firewall(pub-nic-dns), prefix: "none" (in: eth0 66.249.66.127(00:18:f5:63:84:7c):46293 -> 140.105.63.158(00:0f:07:cb:10:91):53 UDP len:68 ttl:42)

Jan 14 06:23:28: DROP service 68->67(udp) from 216.34.233.123 to 217.70.226.162, prefix: "spoof iana-0/8" (in: eth0 217.70.30.115(cc:b2:55:30:fd:ff):68 -> 69.43.103.96(00:03:10:58:1c:f9):67 UDP len:328 ttl:64)

https://regex101.com/



Useful tools

- Text editer (notepad++, sublime, …)
- With unix tools (e.g., sed, awk, vim) https://www.slideshare.net/stevenkim773/regular-expression-regex-vim
- Regex rename