String matching

Data Structures and Algorithms for Computational Linguistics III (ISCL-BA-07)

Çağrı Çöltekin
ccoltekin@sfs.uni-tuebingen.de

University of Tübingen
Seminar für Sprachwissenschaft

Winter Semester 2023/24

version: 3317670 @2023-12-11

Finding patterns in a string

- Finding a pattern in a larger text is a common problem in many applications
- Typical example is searching in a text editor or word processor
- There are many more:
  - DNA sequencing / bioinformatics
  - Plagiarism detection
  - Search engines / information retrieval
  - Spell checking
  - ...

Types of problems

- The efficiency and usability of algorithms depend on some properties of the problem
- Typical applications are based on finding multiple occurrences of a single pattern in a text, where the pattern is much shorter than the text
- The efficiency of the algorithms may depend on the
  - relative size of the pattern
  - expected number of repetitions
  - size of the alphabet
  - whether the pattern is used once or many times
- Another related problem is searching for multiple patterns at once
- In some cases, fuzzy / approximate search may be required
- In some applications, preprocessing (indexing) the text to be searched may be beneficial

Problem definition

and some terminology

text:	A	A	T	A	G	A	C	G	G	C	T	A	G	C	A	A
pattern:				A	G	A	C									

- We want to find all occurrences of pattern p (length m) in text t (length n)
- The characters in both t and p are from an alphabet Σ, in the example $\Sigma = \{A, C, G, T\}$
- The size of the alphabet (q) is often an important factor
- p occurs in t with shift s if $p[0:m] == t[s:s+m]$, we have a match at $s = 3$ in the example
- A string x is a prefix of string y, if $y = xw$ for a possibly empty string w
- A string x is a suffix of string y, if $y = wx$ for a possibly empty string w

Brute-force string search

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
t:	A	A	T	A	G	A	C	G	G	C	T	A	G	C	A	A
p:	A	G	C	A												
j	0	1	2	3												

- Start from the beginning, of $i = 0$ and $j = 0$
  - if $j == m$, announce success with $s = i$
  - if $t[i] != p[j]$: shift p (increase i, set $j = 0$)
  - otherwise: compare the next character (increase i and j, repeat)

Brute-force string search

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
t:	A	A	T	A	G	A	C	G	G	C	T	A	G	C	A	A
p:	A	G	C	A												
j	0	1	2	3												

- Start from the beginning, of $i = 0$ and $j = 0$
  - if $j == m$, announce success with $s = i$
  - if $t[i] != p[j]$: shift p (increase i, set $j = 0$)
  - otherwise: compare the next character (increase i and j, repeat)

Brute-force string search

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
t:	A	A	T	A	G	A	C	G	G	C	T	A	G	C	A	A
p:		A	G	C	A											
j		0	1	2	3											

- Start from the beginning, of $i = 0$ and $j = 0$
  - if $j == m$, announce success with $s = i$
  - if $t[i] != p[j]$: shift p (increase i, set $j = 0$)
  - otherwise: compare the next character (increase i and j, repeat)

Brute-force string search

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
t:	A	A	T	A	G	A	C	G	G	C	T	A	G	C	A	A
p:		A	G	C	A											
j		0	1	2	3											

- Start from the beginning, of $i = 0$ and $j = 0$
  - if $j == m$, announce success with $s = i$
  - if $t[i] != p[j]$: shift p (increase i, set $j = 0$)
  - otherwise: compare the next character (increase i and j, repeat)

Brute-force string search

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
t:	A	A	T	A	G	A	C	G	G	C	T	A	G	C	A	A
p:			A	G	C	A										
j			0	1	2	3										

- Start from the beginning, of $i = 0$ and $j = 0$
  - if $j == m$, announce success with $s = i$
  - if $t[i] != p[j]$: shift p (increase i, set $j = 0$)
  - otherwise: compare the next character (increase i and j, repeat)

Brute-force string search

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
t:	A	A	T	A	G	A	C	G	G	C	T	A	G	C	A	A
p:				A	G	C	A									
j				0	1	2	3									

- Start from the beginning, of $i = 0$ and $j = 0$
  - if $j == m$, announce success with $s = i$
  - if $t[i] != p[j]$: shift p (increase i, set $j = 0$)
  - otherwise: compare the next character (increase i and j, repeat)

Brute-force string search

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
t:	A	A	T	A	G	A	C	G	G	C	T	A	G	C	A	A
p:				A	G	C	A									
j				0	1	2	3									

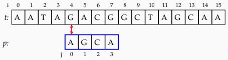
- Start from the beginning, of $i = 0$ and $j = 0$
  - if $j == m$, announce success with $s = i$
  - if $t[i] != p[j]$: shift p (increase i, set $j = 0$)
  - otherwise: compare the next character (increase i and j, repeat)

Brute-force string search

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
t:	A	A	T	A	G	A	C	G	G	C	T	A	G	C	A	A
p:				A	G	C	A									
j				0	1	2	3									

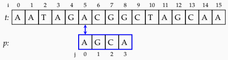
- Start from the beginning, of $i = 0$ and $j = 0$
  - if $j == m$, announce success with $s = i$
  - if $t[i] != p[j]$: shift p (increase i, set $j = 0$)
  - otherwise: compare the next character (increase i and j, repeat)

Brute-force string search



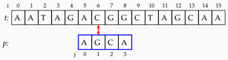
- Start from the beginning, of i = 0 and j = 0
  - if j == m, announce success with s = i
  - if t[i]! = p[j]: shift p (increase i, set j = 0)
  - otherwise: compare the next character (increase i and j, repeat)

Brute-force string search



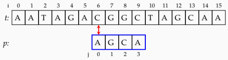
- Start from the beginning, of i = 0 and j = 0
  - if j == m, announce success with s = i
  - if t[i]! = p[j]: shift p (increase i, set j = 0)
  - otherwise: compare the next character (increase i and j, repeat)

Brute-force string search



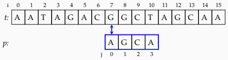
- Start from the beginning, of i = 0 and j = 0
  - if j == m, announce success with s = i
  - if t[i]! = p[j]: shift p (increase i, set j = 0)
  - otherwise: compare the next character (increase i and j, repeat)

Brute-force string search



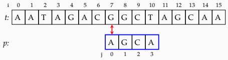
- Start from the beginning, of i = 0 and j = 0
  - if j == m, announce success with s = i
  - if t[i]! = p[j]: shift p (increase i, set j = 0)
  - otherwise: compare the next character (increase i and j, repeat)

Brute-force string search



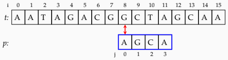
- Start from the beginning, of i = 0 and j = 0
  - if j == m, announce success with s = i
  - if t[i]! = p[j]: shift p (increase i, set j = 0)
  - otherwise: compare the next character (increase i and j, repeat)

Brute-force string search



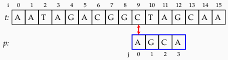
- Start from the beginning, of i = 0 and j = 0
  - if j == m, announce success with s = i
  - if t[i]! = p[j]: shift p (increase i, set j = 0)
  - otherwise: compare the next character (increase i and j, repeat)

Brute-force string search



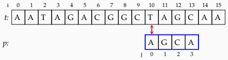
- Start from the beginning, of i = 0 and j = 0
  - if j == m, announce success with s = i
  - if t[i]! = p[j]: shift p (increase i, set j = 0)
  - otherwise: compare the next character (increase i and j, repeat)

Brute-force string search



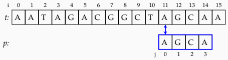
- Start from the beginning, of i = 0 and j = 0
  - if j == m, announce success with s = i
  - if t[i]! = p[j]: shift p (increase i, set j = 0)
  - otherwise: compare the next character (increase i and j, repeat)

Brute-force string search



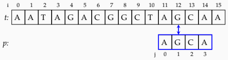
- Start from the beginning, of i = 0 and j = 0
  - if j == m, announce success with s = i
  - if t[i]! = p[j]: shift p (increase i, set j = 0)
  - otherwise: compare the next character (increase i and j, repeat)

Brute-force string search



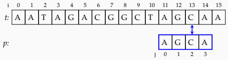
- Start from the beginning, of i = 0 and j = 0
  - if j == m, announce success with s = i
  - if t[i]! = p[j]: shift p (increase i, set j = 0)
  - otherwise: compare the next character (increase i and j, repeat)

Brute-force string search



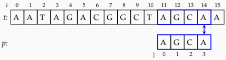
- Start from the beginning, of i = 0 and j = 0
  - if j == m, announce success with s = i
  - if t[i]! = p[j]: shift p (increase i, set j = 0)
  - otherwise: compare the next character (increase i and j, repeat)

Brute-force string search



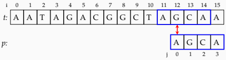
- Start from the beginning, of i = 0 and j = 0
  - if j == m, announce success with s = i
  - if t[i]! = p[j]: shift p (increase i, set j = 0)
  - otherwise: compare the next character (increase i and j, repeat)

Brute-force string search



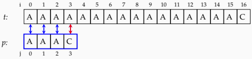
- Start from the beginning, of $i = 0$ and $j = 0$
  - if $j == m$, announce success with $s = i$
  - if $t[i] != p[j]$: shift p (increase i, set $j = 0$)
  - otherwise: compare the next character (increase i and j, repeat)

Brute-force string search

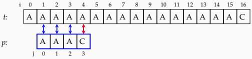


- Start from the beginning, of $i = 0$ and $j = 0$
  - if $j == m$, announce success with $s = i$
  - if $t[i] != p[j]$: shift p (increase i, set $j = 0$)
  - otherwise: compare the next character (increase i and j, repeat)

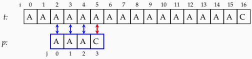
Brute-force approach: worst case



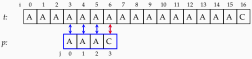
Brute-force approach: worst case



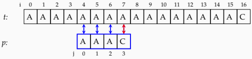
Brute-force approach: worst case



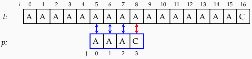
Brute-force approach: worst case



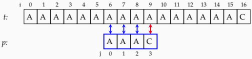
Brute-force approach: worst case



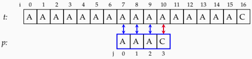
Brute-force approach: worst case



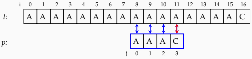
Brute-force approach: worst case



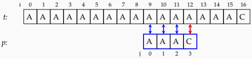
Brute-force approach: worst case



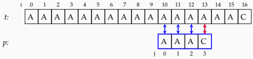
Brute-force approach: worst case



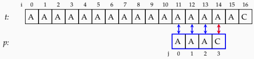
Brute-force approach: worst case



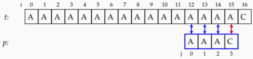
Brute-force approach: worst case



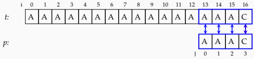
Brute-force approach: worst case



Brute-force approach: worst case



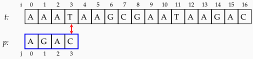
Brute-force approach: worst case



- Worst-case complexity of the method is $O(nm)$
- Crucially, most of the comparisons are redundant
  - for $i > 0$ and any comparison with $j = 0, 1, 2$, we already inspected corresponding i values
- The main idea for more advanced algorithms is to avoid this unnecessary comparisons, with help of additional pre-processing and memory

Boyer-Moore algorithm

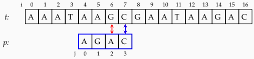
slightly simplified version



- The main idea is to start comparing from the end of p
- If t[i] does not occur in p, shift m steps
- Otherwise, align the last occurrence of t[i] in p with t[i]

Boyer-Moore algorithm

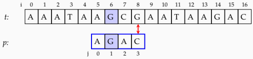
slightly simplified version



- The main idea is to start comparing from the end of p
- If t[i] does not occur in p, shift m steps
- Otherwise, align the last occurrence of t[i] in p with t[i]

Boyer-Moore algorithm

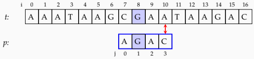
slightly simplified version



- The main idea is to start comparing from the end of p
- If t[i] does not occur in p, shift m steps
- Otherwise, align the last occurrence of t[i] in p with t[i]

Boyer-Moore algorithm

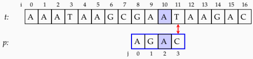
slightly simplified version



- The main idea is to start comparing from the end of p
- If t[i] does not occur in p, shift m steps
- Otherwise, align the last occurrence of t[i] in p with t[i]

Boyer-Moore algorithm

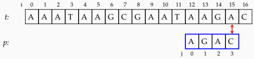
slightly simplified version



- The main idea is to start comparing from the end of p
- If t[i] does not occur in p, shift m steps
- Otherwise, align the last occurrence of t[i] in p with t[i]

Boyer-Moore algorithm

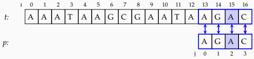
slightly simplified version



- The main idea is to start comparing from the end of p
- If t[i] does not occur in p, shift m steps
- Otherwise, align the last occurrence of t[i] in p with t[i]

Boyer-Moore algorithm

slightly simplified version



- The main idea is to start comparing from the end of p
- If t[i] does not occur in p, shift m steps
- Otherwise, align the last occurrence of t[i] in p with t[i]

Boyer-Moore algorithm

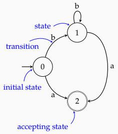
implementation and analysis

- On average, the algorithm performs better than brute-force
- In worst case the complexity of the algorithm is $O(nm)$, example: $t = aaa \dots a, p = baa \dots a$
- Faster versions exist ($O(n + m + q)$)

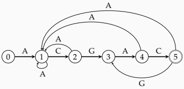
```
last = {}
for j in range(m):
    last[P[j]] = j
i,j = m-1, m-1
while i < n:
    if T[i] == P[j]:
        if j == 0:
            return i
        else:
            i -= 1
            j -= 1
    else:
        k = last.get(T[i], -1)
        i += m - min(j, k+1)
        j = m - 1
return None
```

A quick introduction to FSA

- Another efficient way to search a string is building a finite state automaton for the pattern
- An FSA is a directed graph where edges have labels
- One of the states is the *initial state*
- Some states are accepting states
- We will study FSA more in-depth soon



An FSA for the pattern ACGAC

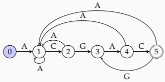


- Start at state 0, switch states based on the input
- All unspecified transitions go to state 0
- When at the accepting state, announce success

FSA pattern matching

demonstration

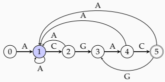
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	A	A	C	G	A	C	G	A	C	A	T	A	C	G	A	C



FSA pattern matching

demonstration

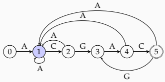
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	A	A	C	G	A	C	G	A	C	A	T	A	C	G	A	C



FSA pattern matching

demonstration

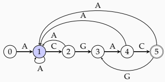
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	A	A	C	G	A	C	G	A	C	A	T	A	C	G	A	C



FSA pattern matching

demonstration

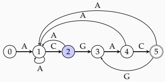
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	A	A	C	G	A	C	G	A	C	A	T	A	C	G	A	C



FSA pattern matching

demonstration

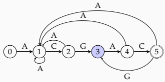
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	A	A	C	G	A	C	G	A	C	A	T	A	C	G	A	C



FSA pattern matching

demonstration

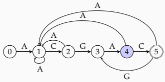
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	A	A	C	G	A	C	G	A	C	A	T	A	C	G	A	C



FSA pattern matching

demonstration

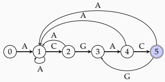
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	A	A	C	G	A	C	G	A	C	A	T	A	C	G	A	C



FSA pattern matching

demonstration

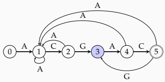
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	A	A	C	G	A	C	G	A	C	A	T	A	C	G	A	C



FSA pattern matching

demonstration

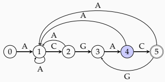
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	A	A	C	G	A	C	G	A	C	A	T	A	C	G	A	C



FSA pattern matching

demonstration

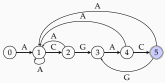
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	A	A	C	G	A	C	G	A	C	A	T	A	C	G	A	C



FSA pattern matching

demonstration

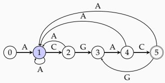
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	A	A	C	G	A	C	G	A	C	A	T	A	C	G	A	C



FSA pattern matching

demonstration

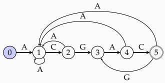
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	A	A	C	G	A	C	G	A	C	A	T	A	C	G	A	C



FSA pattern matching

demonstration

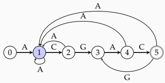
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	A	A	C	G	A	C	G	A	C	A	T	A	C	G	A	C



FSA pattern matching

demonstration

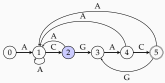
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	A	A	C	G	A	C	G	A	C	A	T	A	C	G	A	C



FSA pattern matching

demonstration

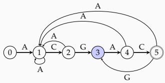
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	A	A	C	G	A	C	G	A	C	A	T	A	C	G	A	C



FSA pattern matching

demonstration

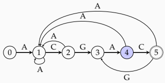
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	A	A	C	G	A	C	G	A	C	A	T	A	C	G	A	C



FSA pattern matching

demonstration

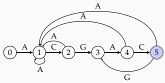
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	A	A	C	G	A	C	G	A	C	A	T	A	C	G	A	C



FSA pattern matching

demonstration

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	A	A	C	G	A	C	G	A	C	A	T	A	C	G	A	C



FSA for string matching

how to build the automaton

- An FSA results in $O(n)$ time matching, however, we need to first build the automaton
- At any state of the automaton, we want to know which state to go for the failing matches
- Given substring s recognized by a state and a non-matching input symbol a, we want to find the longest prefix of s such that it is also a suffix of sa
- A naive attempt results in $O(qm^3)$ time for building the automaton (where q is the size of the alphabet m is the length of the pattern)
- If stored in a matrix, the space requirement is $O(qm)$
- Better (faster) algorithms exist for construction these automaton (we will cover some later in this course)

Knuth-Morris-Pratt (KMP) algorithm

- The KMP algorithm is probably the most popular algorithm for string matching
- The idea is similar to the FSA approach: on failure, continue comparing from the longest matched prefix so far
- However, we rely on a simpler data structure (a function/table that tells us where to back up)
- Construction of the table is also faster

KMP algorithm

demonstration

i:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
t:	A	A	C	G	A	T	G	A	C	A	T	A	C	G	A	C	A	T	G

p:	A	C	G	A	C
j:	0	1	2	3	4
f:	0	0	0	1	2

- In case of a match, increment both i and j
- On failure, or at the end of the pattern, decide which new $p[j]$ compare with $t[i]$ based on a function f
- $f[j-1]$ tells which j value to resume the comparisons from

KMP algorithm

demonstration

i:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
t:	A	A	C	G	A	T	G	A	C	A	T	A	C	G	A	C	A	T	G

p:	A	C	G	A	C
j:	0	1	2	3	4
f:	0	0	0	1	2

- In case of a match, increment both i and j
- On failure, or at the end of the pattern, decide which new $p[j]$ compare with $t[i]$ based on a function f
- $f[j-1]$ tells which j value to resume the comparisons from

KMP algorithm

demonstration

i:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
t:	A	A	C	G	A	T	G	A	C	A	T	A	C	G	A	C	A	T	G

p:	A	C	G	A	C
j:	0	1	2	3	4
f:	0	0	0	1	2

- In case of a match, increment both i and j
- On failure, or at the end of the pattern, decide which new p[j] compare with t[i] based on a function f
- f[j − 1] tells which j value to resume the comparisons from

KMP algorithm

demonstration

i:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
t:	A	A	C	G	A	T	G	A	C	A	T	A	C	G	A	C	A	T	G

p:	A	C	G	A	C
j:	0	1	2	3	4
f:	0	0	0	1	2

- In case of a match, increment both i and j
- On failure, or at the end of the pattern, decide which new p[j] compare with t[i] based on a function f
- f[j − 1] tells which j value to resume the comparisons from

KMP algorithm

demonstration

i:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
t:	A	A	C	G	A	T	G	A	C	A	T	A	C	G	A	C	A	T	G

p:	A	C	G	A	C
j:	0	1	2	3	4
f:	0	0	0	1	2

- In case of a match, increment both i and j
- On failure, or at the end of the pattern, decide which new p[j] compare with t[i] based on a function f
- f[j − 1] tells which j value to resume the comparisons from

KMP algorithm

demonstration

i:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
t:	A	A	C	G	A	T	G	A	C	A	T	A	C	G	A	C	A	T	G

p:	A	C	G	A	C
j:	0	1	2	3	4
f:	0	0	0	1	2

- In case of a match, increment both i and j
- On failure, or at the end of the pattern, decide which new p[j] compare with t[i] based on a function f
- f[j − 1] tells which j value to resume the comparisons from

KMP algorithm

demonstration

i:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
t:	A	A	C	G	A	T	G	A	C	A	T	A	C	G	A	C	A	T	G

p:	A	C	G	A	C
j:	0	1	2	3	4
f:	0	0	0	1	2

- In case of a match, increment both i and j
- On failure, or at the end of the pattern, decide which new p[j] compare with t[i] based on a function f
- f[j − 1] tells which j value to resume the comparisons from

KMP algorithm

demonstration

i:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
t:	A	A	C	G	A	T	G	A	C	A	T	A	C	G	A	C	A	T	G

p:	A	C	G	A	C
j:	0	1	2	3	4
f:	0	0	0	1	2

- In case of a match, increment both i and j
- On failure, or at the end of the pattern, decide which new p[j] compare with t[i] based on a function f
- f[j − 1] tells which j value to resume the comparisons from

KMP algorithm

demonstration

i:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
t:	A	A	C	G	A	T	G	A	C	A	T	A	C	G	A	C	A	T	G

p:	A	C	G	A	C
j:	0	1	2	3	4
f:	0	0	0	1	2

- In case of a match, increment both i and j
- On failure, or at the end of the pattern, decide which new p[j] compare with t[i] based on a function f
- f[j − 1] tells which j value to resume the comparisons from

KMP algorithm

demonstration

i:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
t:	A	A	C	G	A	T	G	A	C	A	T	A	C	G	A	C	A	T	G

p:	A	C	G	A	C
j:	0	1	2	3	4
f:	0	0	0	1	2

- In case of a match, increment both i and j
- On failure, or at the end of the pattern, decide which new p[j] compare with t[i] based on a function f
- f[j − 1] tells which j value to resume the comparisons from

Complexity of the KMP algorithm

- In the while loop, we either increase i, or shift the comparison
- As a result, the loop runs at most $2n$ times, complexity is $O(n)$

```
i, j = 0, 0
while i < n:
  if T[i] == P[j]:
    if j == m - 1:
      return i
    else:
      i += 1
      j += 1
  elif j > 0:
    j = f[j - 1]
  else:
    i += 1
return None
```


Building the prefix/failure table

```
f = [0] * m
j, k = 1, 0
while j < m:
  if P[j] == P[k]:
    f[j] = k + 1
    j += 1
    k += 1
  elif k > 0:
    k = f[k - 1]
  else:
    j += 1
```

j

p:	A	T	C	A	C	A	T	C	A	T	T
f:	0	0	0	0	0	0	0	0	0	0	0

k

Building the prefix/failure table

```
f = [0] * m
j, k = 1, 0
while j < m:
  if P[j] == P[k]:
    f[j] = k + 1
    j += 1
    k += 1
  elif k > 0:
    k = f[k - 1]
  else:
    j += 1
```

j

p:	A	T	C	A	C	A	T	C	A	T	T
f:	0	0	0	0	0	0	0	0	0	0	0

k

Building the prefix/failure table

```
f = [0] * m
j, k = 1, 0
while j < m:
  if P[j] == P[k]:
    f[j] = k + 1
    j += 1
    k += 1
  elif k > 0:
    k = f[k - 1]
  else:
    j += 1
```

j

p:	A	T	C	A	C	A	T	C	A	T	T
f:	0	0	0	1	0	0	0	0	0	0	0

k

Building the prefix/failure table

```
f = [0] * m
j, k = 1, 0
while j < m:
  if P[j] == P[k]:
    f[j] = k + 1
    j += 1
    k += 1
  elif k > 0:
    k = f[k - 1]
  else:
    j += 1
```

					j						
p:	A	T	C	A	C	A	T	C	A	T	T
		k									
f:	0	0	0	1	0	0	0	0	0	0	0

Building the prefix/failure table

```
f = [0] * m
j, k = 1, 0
while j < m:
  if P[j] == P[k]:
    f[j] = k + 1
    j += 1
    k += 1
  elif k > 0:
    k = f[k - 1]
  else:
    j += 1
```

					j						
p:	A	T	C	A	C	A	T	C	A	T	T
	k										
f:	0	0	0	1	0	0	0	0	0	0	0

Building the prefix/failure table

```
f = [0] * m
j, k = 1, 0
while j < m:
  if P[j] == P[k]:
    f[j] = k + 1
    j += 1
    k += 1
  elif k > 0:
    k = f[k - 1]
  else:
    j += 1
```

						j					
p:	A	T	C	A	C	A	T	C	A	T	T
	k										
f:	0	0	0	1	0	1	0	0	0	0	0

Building the prefix/failure table

```
f = [0] * m
j, k = 1, 0
while j < m:
  if P[j] == P[k]:
    f[j] = k + 1
    j += 1
    k += 1
  elif k > 0:
    k = f[k - 1]
  else:
    j += 1
```

							j				
p:	A	T	C	A	C	A	T	C	A	T	T
		k									
f:	0	0	0	1	0	1	2	0	0	0	0

Building the prefix/failure table

```
f = [0] * m
j, k = 1, 0
while j < m:
  if P[j] == P[k]:
    f[j] = k + 1
    j += 1
    k += 1
  elif k > 0:
    k = f[k - 1]
  else:
    j += 1
```

								j			
p:	A	T	C	A	C	A	T	C	A	T	T
			k								
f:	0	0	0	1	0	1	2	3	0	0	0

Building the prefix/failure table

```
f = [0] * m
j, k = 1, 0
while j < m:
  if P[j] == P[k]:
    f[j] = k + 1
    j += 1
    k += 1
  elif k > 0:
    k = f[k - 1]
  else:
    j += 1
```

									j		
p:	A	T	C	A	C	A	T	C	A	T	T
				k							
f:	0	0	0	1	0	1	2	3	4	0	0

Building the prefix/failure table

```
f = [0] * m
j, k = 1, 0
while j < m:
  if P[j] == P[k]:
    f[j] = k + 1
    j += 1
    k += 1
  elif k > 0:
    k = f[k - 1]
  else:
    j += 1
```

										j	
p:	A	T	C	A	C	A	T	C	A	T	T
					k						
f:	0	0	0	1	0	1	2	3	4	0	0

Building the prefix/failure table

```
f = [0] * m
j, k = 1, 0
while j < m:
  if P[j] == P[k]:
    f[j] = k + 1
    j += 1
    k += 1
  elif k > 0:
    k = f[k - 1]
  else:
    j += 1
```

										j	
p:	A	T	C	A	C	A	T	C	A	T	T
		k									
f:	0	0	0	1	0	1	2	3	4	2	0

Building the prefix/failure table

```
f = [0] * m
j, k = 1, 0
while j < m:
  if P[j] == P[k]:
    f[j] = k + 1
    j += 1
    k += 1
  elif k > 0:
    k = f[k - 1]
  else:
    j += 1
```

										j	
p:	A	T	C	A	C	A	T	C	A	T	T
			k								
f:	0	0	0	1	0	1	2	3	4	2	0

Building the prefix/failure table

```
f = [0] * m
j, k = 1, 0
while j < m:
  if P[j] == P[k]:
    f[j] = k + 1
    j += 1
    k += 1
  elif k > 0:
    k = f[k - 1]
  else:
    j += 1
```

										j	
p:	A	T	C	A	C	A	T	C	A	T	T
	k										
f:	0	0	0	1	0	1	2	3	4	2	0

Building the prefix/failure table

```
f = [0] * m
j, k = 1, 0
while j < m:
  if P[j] == P[k]:
    f[j] = k + 1
    j += 1
    k += 1
  elif k > 0:
    k = f[k - 1]
  else:
    j += 1
```

											j
p:	A	T	C	A	C	A	T	C	A	T	T
	k										
f:	0	0	0	1	0	1	2	3	4	2	0

Rabin-Karp algorithm

- Rabin-Karp string matching algorithm is another interesting algorithm
- The idea is instead of matching the string itself, matching the hash of it (based on a hash function)
- If a match found, we need to verify – the match may be because of a hash collision
- Otherwise, the algorithm makes a single comparison for each position in the text
- However, a hash should be computed for each position (with size m)
- Rolling hash functions avoid this complication

Rabin-Karp string matching

demonstration with additive hashing

t:	7	1	3	6	7	4	3	8	5	7	9	4	3	9

$h = 39$

p:	4	3	8	5	7	9	4	3

$h(p) = 43$

- A rolling hash function changes the hash value only based on the item coming in and going out of the window
- To reduce collisions, better rolling-hash functions (e.g., polynomial hash functions) can also be used

Rabin-Karp string matching

demonstration with additive hashing

t:	7	1	3	6	7	4	3	8	5	7	9	4	3	9

$h = 37$

p:	4	3	8	5	7	9	4	3

$h(p) = 43$

- A rolling hash function changes the hash value only based on the item coming in and going out of the window
- To reduce collisions, better rolling-hash functions (e.g., polynomial hash functions) can also be used

Rabin-Karp string matching

demonstration with additive hashing

?

t:	7	1	3	6	7	4	3	8	5	7	9	4	3	9

$h = 43$

p:	4	3	8	5	7	9	4	3

$h(p) = 43$

- A rolling hash function changes the hash value only based on the item coming in and going out of the window
- To reduce collisions, better rolling-hash functions (e.g., polynomial hash functions) can also be used

Rabin-Karp string matching

demonstration with additive hashing

t:	7	1	3	6	7	4	3	8	5	7	9	4	3	9

$h = 49$

p:	4	3	8	5	7	9	4	3

$h(p) = 43$

- A rolling hash function changes the hash value only based on the item coming in and going out of the window
- To reduce collisions, better rolling-hash functions (e.g., polynomial hash functions) can also be used

Rabin-Karp string matching

demonstration with additive hashing

t:	7	1	3	6	7	4	3	8	5	7	9	4	3	9

$h = 47$

p:	4	3	8	5	7	9	4	3

$h(p) = 43$

- A rolling hash function changes the hash value only based on the item coming in and going out of the window
- To reduce collisions, better rolling-hash functions (e.g., polynomial hash functions) can also be used

Rabin-Karp string matching

demonstration with additive hashing

?

t:	7	1	3	6	7	4	3	8	5	7	9	4	3	9

$h = 43$

p:	4	3	8	5	7	9	4	3

$h(p) = 43$

- A rolling hash function changes the hash value only based on the item coming in and going out of the window
- To reduce collisions, better rolling-hash functions (e.g., polynomial hash functions) can also be used

Rabin-Karp string matching

demonstration with additive hashing

t:	7	1	3	6	7	4	3	8	5	7	9	4	3	9

$h = 48$

p:	4	3	8	5	7	9	4	3

$h(p) = 43$

- A rolling hash function changes the hash value only based on the item coming in and going out of the window
- To reduce collisions, better rolling-hash functions (e.g., polynomial hash functions) can also be used

Summary

- String matching is an important problem with wide range of applications
- The choice of algorithm largely depends on the problem
- We will revisit the problem on regular expressions and finite-state automata
- Reading: Goodrich, Tamassia, and Goldwasser (2013, chapter 13)

Next:

- Algorithms on strings: edit distance / alignment
- Reading: Goodrich, Tamassia, and Goldwasser (2013, chapter 13),Jurafsky and Martin (2009, section 3.11, or 2.5 in online draft)

Building the prefix/failure table

another example

```
f = [0] * m
j, k = 1, 0
while j < m:
  if P[j] == P[k]:
    f[j] = k + 1
    j += 1
    k += 1
  elif k > 0:
    k = f[k - 1]
  else:
    j += 1
```

j

p:	A	T	A	C	G	A	T	A	C	A	T	G	C

k

f:	0	0	0	0	0	0	0	0	0	0	0	0	0

Building the prefix/failure table

another example

```
f = [0] * m
j, k = 1, 0
while j < m:
  if P[j] == P[k]:
    f[j] = k + 1
    j += 1
    k += 1
  elif k > 0:
    k = f[k - 1]
  else:
    j += 1
```

j

p:	A	T	A	C	G	A	T	A	C	A	T	G	C

k

f:	0	0	1	0	0	0	0	0	0	0	0	0	0

Building the prefix/failure table

another example

```
f = [0] * m
j, k = 1, 0
while j < m:
  if P[j] == P[k]:
    f[j] = k + 1
    j += 1
    k += 1
  elif k > 0:
    k = f[k - 1]
  else:
    j += 1
```

j

p:	A	T	A	C	G	A	T	A	C	A	T	G	C

k

f:	0	0	1	0	0	0	0	0	0	0	0	0	0

Building the prefix/failure table

another example

```
f = [0] * m
j, k = 1, 0
while j < m:
  if P[j] == P[k]:
    f[j] = k + 1
    j += 1
    k += 1
  elif k > 0:
    k = f[k - 1]
  else:
    j += 1
```

j

p:	A	T	A	C	G	A	T	A	C	A	T	G	C

k

f:	0	0	1	0	0	0	0	0	0	0	0	0	0

Building the prefix/failure table

another example

```
f = [0] * m
j, k = 1, 0
while j < m:
    if P[j] == P[k]:
        f[j] = k + 1
        j += 1
        k += 1
    elif k > 0:
        k = f[k - 1]
    else:
        j += 1
```

						j							
p:	A	T	A	C	G	A	T	A	C	A	T	G	C
	k												
f:	0	0	1	0	0	1	0	0	0	0	0	0	0

Building the prefix/failure table

another example

```
f = [0] * m
j, k = 1, 0
while j < m:
    if P[j] == P[k]:
        f[j] = k + 1
        j += 1
        k += 1
    elif k > 0:
        k = f[k - 1]
    else:
        j += 1
```

							j						
p:	A	T	A	C	G	A	T	A	C	A	T	G	C
		k											
f:	0	0	1	0	0	1	2	0	0	0	0	0	0

Building the prefix/failure table

another example

```
f = [0] * m
j, k = 1, 0
while j < m:
    if P[j] == P[k]:
        f[j] = k + 1
        j += 1
        k += 1
    elif k > 0:
        k = f[k - 1]
    else:
        j += 1
```

								j					
p:	A	T	A	C	G	A	T	A	C	A	T	G	C
			k										
f:	0	0	1	0	0	1	2	3	0	0	0	0	0

Building the prefix/failure table

another example

```
f = [0] * m
j, k = 1, 0
while j < m:
    if P[j] == P[k]:
        f[j] = k + 1
        j += 1
        k += 1
    elif k > 0:
        k = f[k - 1]
    else:
        j += 1
```

									j				
p:	A	T	A	C	G	A	T	A	C	A	T	G	C
				k									
f:	0	0	1	0	0	1	2	3	4	0	0	0	0

Building the prefix/failure table

another example

```
f = [0] * m
j, k = 1, 0
while j < m:
    if P[j] == P[k]:
        f[j] = k + 1
        j += 1
        k += 1
    elif k > 0:
        k = f[k - 1]
    else:
        j += 1
```

										j			
p:	A	T	A	C	G	A	T	A	C	A	T	G	C
					k								
f:	0	0	1	0	0	1	2	3	4	0	0	0	0

Building the prefix/failure table

another example

```
f = [0] * m
j, k = 1, 0
while j < m:
    if P[j] == P[k]:
        f[j] = k + 1
        j += 1
        k += 1
    elif k > 0:
        k = f[k - 1]
    else:
        j += 1
```

										j			
p:	A	T	A	C	G	A	T	A	C	A	T	G	C
	k												
f:	0	0	1	0	0	1	2	3	4	1	0	0	0

Building the prefix/failure table

another example

```
f = [0] * m
j, k = 1, 0
while j < m:
    if P[j] == P[k]:
        f[j] = k + 1
        j += 1
        k += 1
    elif k > 0:
        k = f[k - 1]
    else:
        j += 1
```

											j		
p:	A	T	A	C	G	A	T	A	C	A	T	G	C
		k											
f:	0	0	1	0	0	1	2	3	4	1	2	0	0

Building the prefix/failure table

another example

```
f = [0] * m
j, k = 1, 0
while j < m:
    if P[j] == P[k]:
        f[j] = k + 1
        j += 1
        k += 1
    elif k > 0:
        k = f[k - 1]
    else:
        j += 1
```

												j	
p:	A	T	A	C	G	A	T	A	C	A	T	G	C
			k										
f:	0	0	1	0	0	1	2	3	4	1	2	0	0

Building the prefix/failure table

another example

```
f = [0] * m
j, k = 1, 0
while j < m:
    if P[j] == P[k]:
        f[j] = k + 1
        j += 1
        k += 1
    elif k > 0:
        k = f[k - 1]
    else:
        j += 1
```

												j	
p:	A	T	A	C	G	A	T	A	C	A	T	G	C
		k											
f:	0	0	1	0	0	1	2	3	4	1	2	0	0

Building the prefix/failure table

another example

```
f = [0] * m
j, k = 1, 0
while j < m:
    if P[j] == P[k]:
        f[j] = k + 1
        j += 1
        k += 1
    elif k > 0:
        k = f[k - 1]
    else:
        j += 1
```

												j	
p:	A	T	A	C	G	A	T	A	C	A	T	G	C
	k												
f:	0	0	1	0	0	1	2	3	4	1	2	0	0

Building the prefix/failure table

another example

```
f = [0] * m
j, k = 1, 0
while j < m:
    if P[j] == P[k]:
        f[j] = k + 1
        j += 1
        k += 1
    elif k > 0:
        k = f[k - 1]
    else:
        j += 1
```

													j
p:	A	T	A	C	G	A	T	A	C	A	T	G	C
	k												
f:	0	0	1	0	0	1	2	3	4	1	2	0	0

Acknowledgments, credits, references

Goodrich, Michael T., Roberto Tamassia, and Michael H. Goldwasser (2013). *Data Structures and Algorithms in Python*. John Wiley & Sons, Incorporated. ISBN: 9781118476734.

Jurafsky, Daniel and James H. Martin (2009). *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. second edition. Pearson Prentice Hall. ISBN: 978-0-13-504196-3.