Pattern-Matching Algorithms

Applied Algorithms

Digitized Text

- Text processing remains one of the dominant functions of computers. Computer are used to edit, store, and display documents.
- New data is being generated at a rapidly increasing pace.
- Examples of digital collections: email archives, customer reviews, documents stored locally on a user's computer, snapshots of the www, social media data, e.g., Twitter, Facebook etc.
- These collections include written text from hundreds of international languages.
- Furthermore, there are large data sets (such as DNA) that can be viewed computationally as "strings" even though they are not language.

The Brute-force Method

- In this part, we will explore some of the fundamental algorithms that can be used to efficiently analyze and process large textual data sets.
- In addition to having interesting applications, text-processing algorithms also highlight some important algorithmic design patterns.
- Searching for a pattern as a substring of a larger piece of text, e.g., a word in a document.
- Our first approach for the pattern-matching problem will be with the brute-force method — often inefficient

Dynamic Programming and The Greedy Methods

- Dynamic programming can be applied in certain settings to solve a problem in polynomial time that appears at first to require exponential time to solve.
- Finding partial matches between strings that may be similar but not perfectly aligned.
- Examples: making suggestions for a misspelled word, matching related genetic samples.
- The greedy method allows us to approximate solutions to hard problems, e.g., text compression.
- We will explore the data structures to better organize textual data.

Notations for Strings and the Python str Class

- Character strings can come from various sources. For example:

$$S = \texttt{"CGTAAACTGCTTTAATCAAACGC"}$$
$$T = \texttt{"http://www.wiley.com"}$$

- We will assume each character in a string is a member of an alphabet, Σ. For example, the alphabet for a DNA strings, Σ = {A,C,G,T}.

- The size of an alphabet, |Σ|, is nontrivial. Unicode alphabet has more than a million distinct characters. |Σ| is important in asymptotic analysis for text-processing algorithms.

Pattern-Matching Algorithms

- Given a text string T of length n and a pattern string P of length m, find whether P is a substring of T. Our goal is to locate the lowest index j within T at which P begins, such that T[j:j+m] equals P. There could be more than one P in T.
- Pattern-matching problem and str class (behaviors):
  - P in T, T.find(P), T.index(P), T.count(P)
  - more complex behaviors: T.partition(P), T.split(P), and T.replace(P, Q).

Brute-Force

- Enumerate all possible configurations of the inputs involved and pick the best among all these configurations.
- In other words, test all the possible placements of P relative to T
- Exhaustive search approach.
- The worst-case running time of the brute-force method is O(nm).

An implementation of brute-force pattern-matching algorithm.

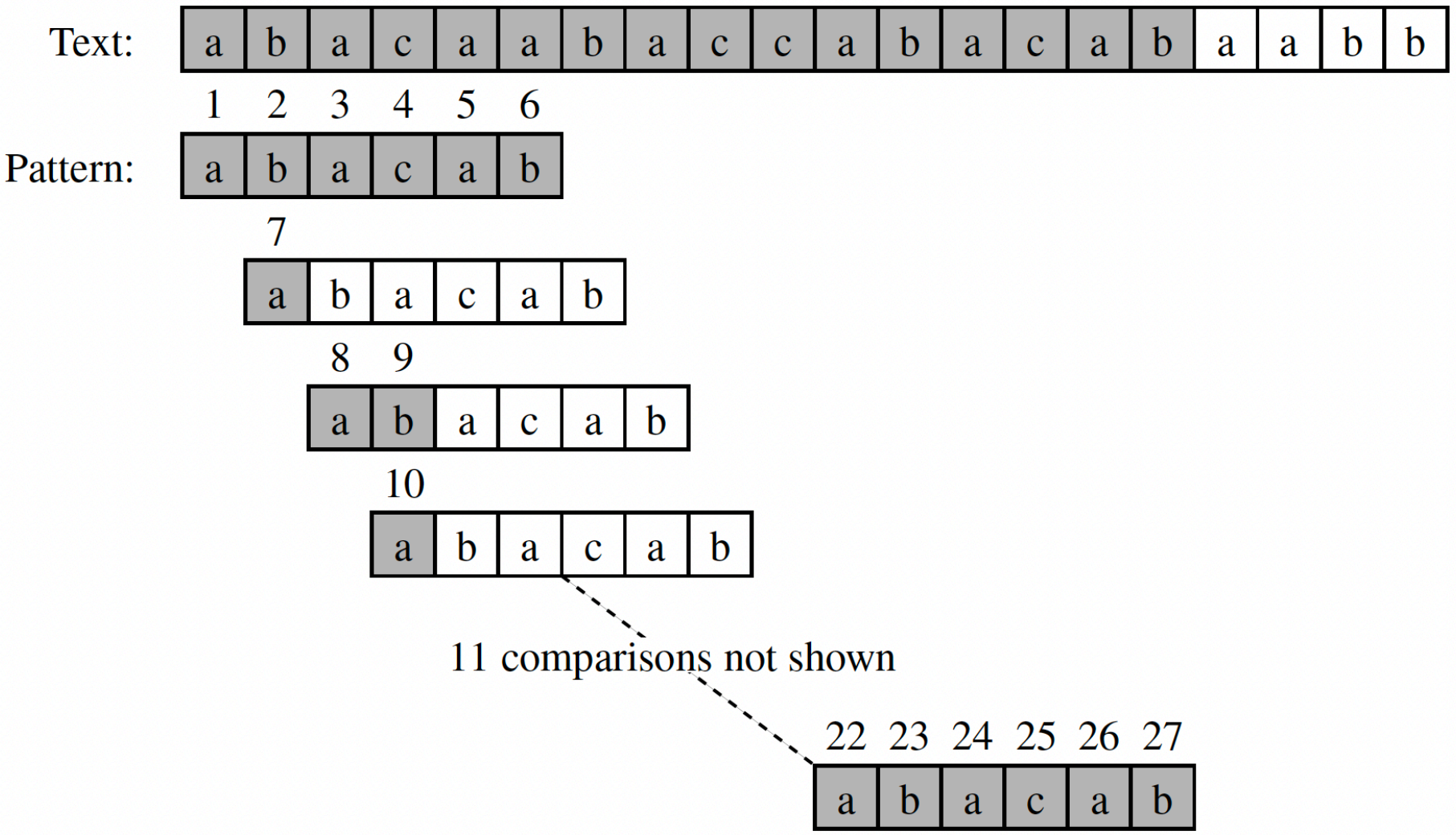
```
1  def find_brute(T, P):
2    """Return the lowest index of T at which substring P begins (or else -1)."""
3    n, m = len(T), len(P)                  # introduce convenient notations
4    for i in range(n−m+1):                 # try every potential starting index within T
5      k = 0                                # an index into pattern P
6      while k < m and T[i + k] == P[k]:        # kth character of P matches
7        k += 1
8      if k == m:                           # if we reached the end of pattern,
9        return i                           # substring T[i:i+m] matches P
10   return −1                              # failed to find a match starting with any i
```

- T = “abacaabaccabacabaabb"
- P = “abacab”

Brute Force: Example

- Suppose we are given the text string and the pattern strings:
  - T = “abacaabaccabacabaabb"
  - P = “abacab”
- Show the execution of the brute-force pattern-matching algorithm on T and P.

Brute Force: Example



The algorithm performs 27 character comparisons

Brute Force: class exercise

- Video: https://www.youtube.com/watch?v=nK7SLhXcqRo
- Class Exercise: How many comparison?
  - T = “abacaabaccabacabaabb"
  - P = “cabac”

Discussion: Pattern Matching Problem

- Example 1:

T: u s b j m u s b j o

P: u s b j o

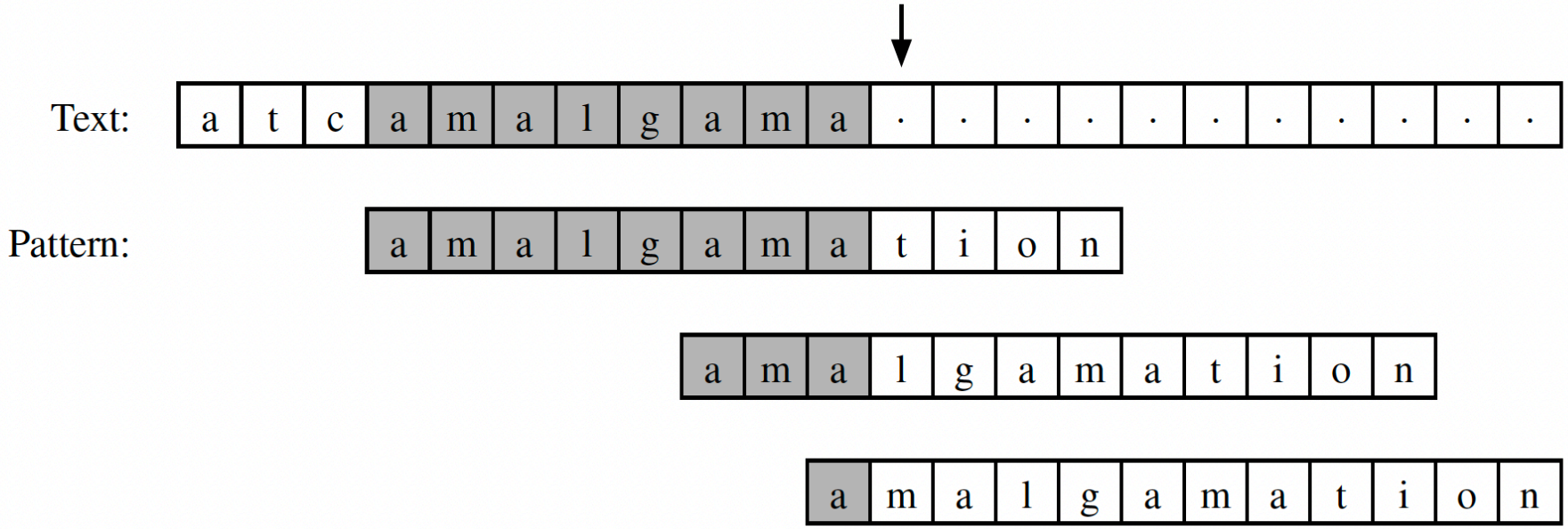
- Example 2:

T: k l m k l m l z r t

P: k l m k l z

Discussion: Pattern Matching Problem

- Example 3:

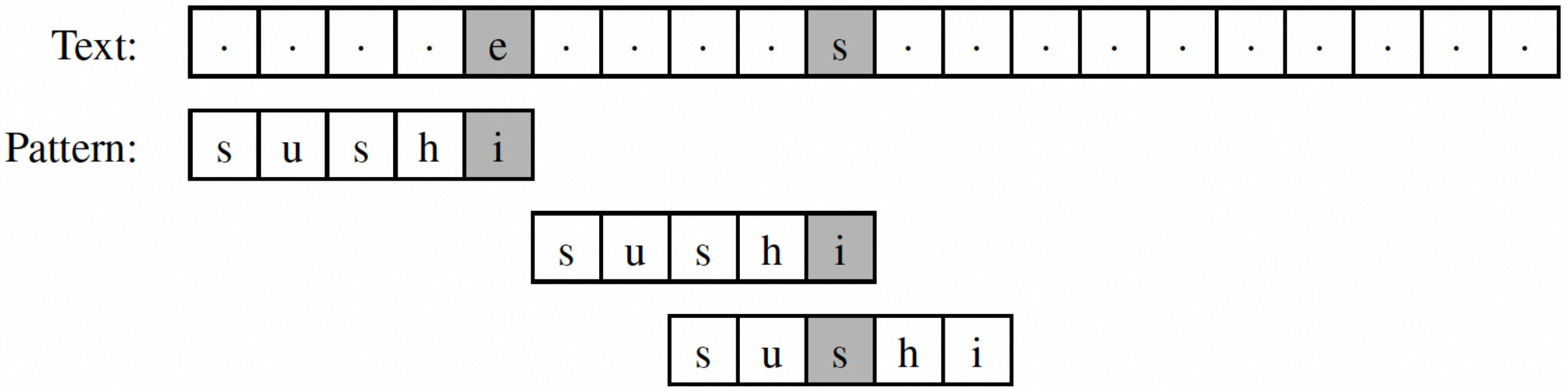


The Boyer-Moore Algorithm

- The Boyer-Moore algorithm improves the running time of the brute-force algorithm by using two heuristics.
  - *Looking-Glass Heuristic:* Start the comparison from the end of P and move backward to the front of P.
  - *Character-Jump Heuristic:* Handle the mismatches as follows:
    - Once a mismatch occurs of text character T[i]=**c** with the corresponding pattern character P[k] is handled as follows:
    - Let T[i]=**c** represent mismatch character while checking P[k]. If **c** does not exist in P, then shift P completely past T[i].
    - Otherwise, shift P until you find a **c** in P gets aligned with T[i]
- We describe a simplified version of the Boyer and Moore algorithm.

The Boyer-Moore Algorithm: Simple Example

- *First mismatch between e-i:* e does not exist in P, the entire P is shifted beyond its location.
- *Second mismatch between s-i:* the mismatched character is in P, the pattern is next shifted so that its last occurrence of s is aligned with the corresponding s in the text.



The Boyer-Moore Algorithm:

- Lookup table (last function): used to find out where a mismatched character occurs elsewhere in the pattern.
- A lookup table can be implemented using a hash table with worst-case O(1) access time. The size of hash table and alphabet is same. There will be time required to initialize the table.
- The space usage depends on distinct pattern characters —O(m).
- The worst-case running time of the Boyer-Moore algorithm is O(nm+|Σ|).
- In the worst case, a look up table takes O(m+|Σ|) time and the search for the pattern takes O(nm) time.

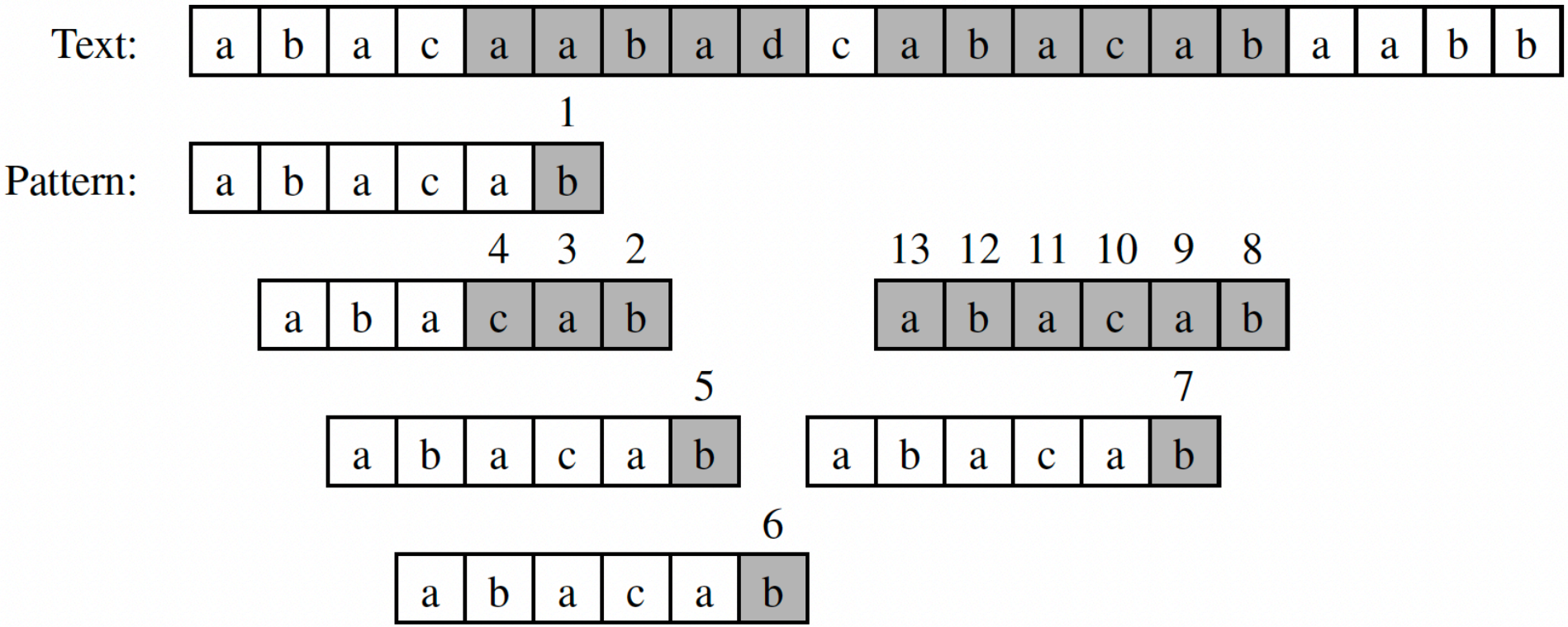
The Boyer-Moore Algorithm: Implementation

```
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 Boyer-Moore Algorithm: Example

- Last(c) is the location of the last occurrence of c in P.

c	a	b	c	d
last(c)	4	5	3	−1



The Knuth-Morris-Pratt Algorithm

- Major inefficiency with the worst-case performances of the brute-force and Boyer-Moore pattern-matching algorithms: When there are several matching characters but then detect a mismatch, the algorithms ignore all the information gained by the successful comparisons.

- The Knuth-Morris-Pratt (KMP) algorithm does not ignore successful matches once a mismatch happens and find P in T in O(n+m) time.

- How?

  - Precompute a failure function, f, to know self-overlaps between portions of the pattern and use f to handle mismatches.

  - KMP uses f for better shifts once a mismatch occurs.

The Knuth-Morris-Pratt Algorithm: The Failure Function

- P = “amalgamation”. The Knuth-Morris-Pratt (KMP) failure function, f (k), for the string P is

k	0	1	2	3	4	5	6	7	8	9	10	11
$P[k]$	a	m	a	l	g	a	m	a	t	i	o	n
$f(k)$	0	0	1	0	0	1	2	3	0	0	0	0

The Knuth-Morris-Pratt Algorithm: 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
```

□

The Knuth-Morris-Pratt Algorithm: Constructing the KMP Failure Function

```
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
```

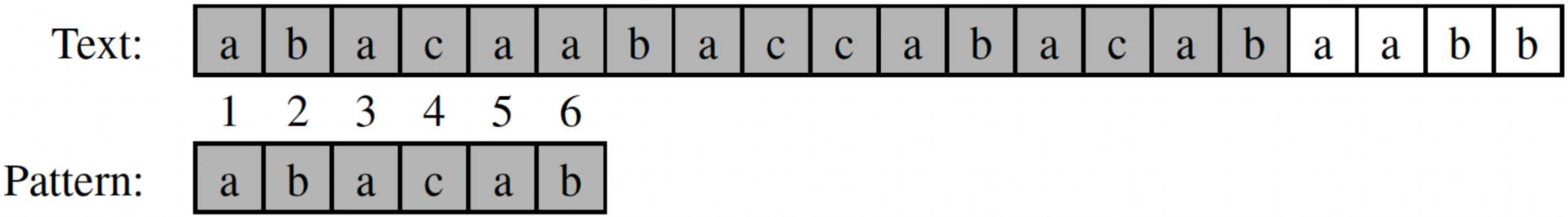
Class Exercise

- Exercise :

T: k l m k l m k l z r t

P: k l m k l z

- Exercise :



References

1. Hetland, M. L. (2014). Python Algorithms: mastering basic algorithms in the Python Language. Apress.
2. Lee, K. D., Lee, K. D., & Steve Hubbard, S. H. (2015). *Data Structures and Algorithms with Python*. Springer.
3. Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2013). *Data structures and algorithms in Python*. Hoboken: Wiley
4. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms*. MIT press.
5. Sedgewick, R., & Wayne, K. (2011). *Algorithms*. Addison-wesley professional.