IT309: Text Processing – Brute Force

TEXT PROCESSING

The world is awash in text data

- Text processing is one of the dominant functions of computers
- Being able to search text archives is a very common activity

Examples:

- Electronically stored documents – private and public
- Email archives
- WWW data, including search histories

But also:

- DNA sequences: “CGTAAACTGCTTAATCAAACGC”
- URLs: http://www.wiley.com

STRINGS

- In Python text is represented with the **string** data type
- A **string** is a sequence of characters

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

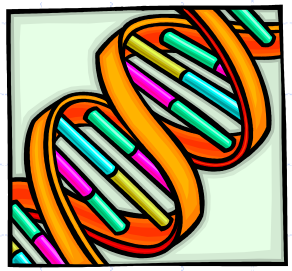
Example of alphabets:

- ASCII
- Unicode
- {0, 1}
- {A, C, G, T}

String Pattern Matching

- 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 positions 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

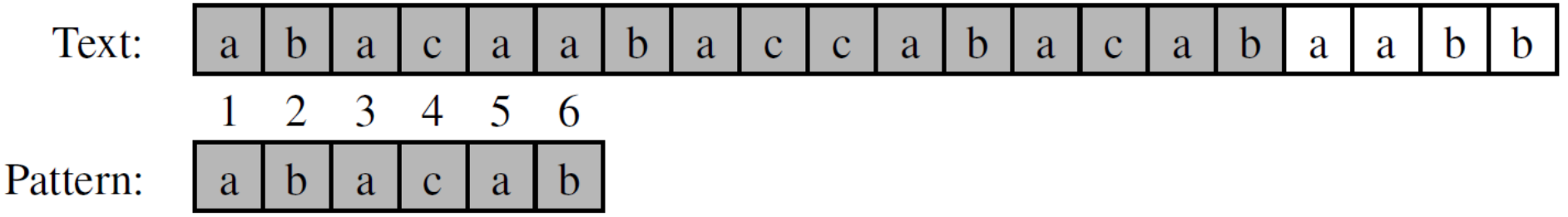
Brute-Force Pattern Matching



- 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 … ah***
  - ***P*** = ***aaah***
  - may occur in images and DNA sequences
  - unlikely in English text

BRUTE FORCE PATTERN MATCHING

The basic pattern matching problem is to find whether a given string pattern (P) occurs in a larger body of text (P).

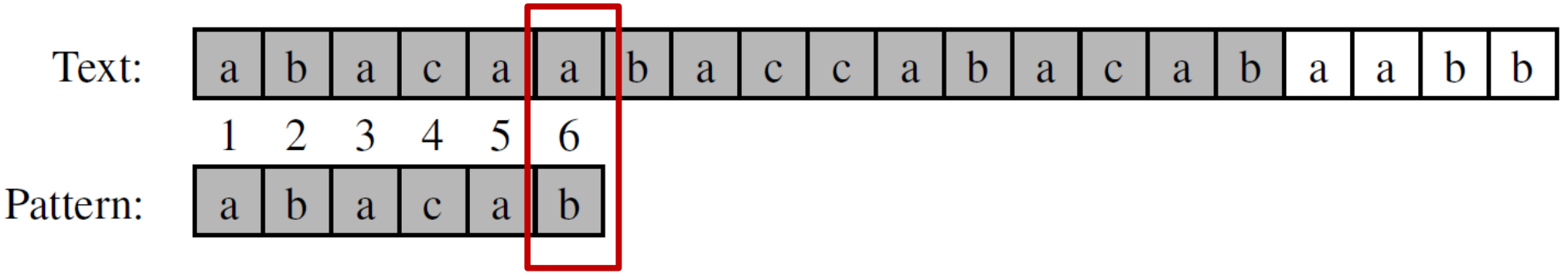


- Line up the pattern's first character with the first character of the text
- Check each character pair
- If equal, check to see whether the next characters of T and P match
- Continue checking until we discover all characters of P match those in T or we encounter a mismatch

- Note: The numbers above the cells indicate the sequence of character comparisons.

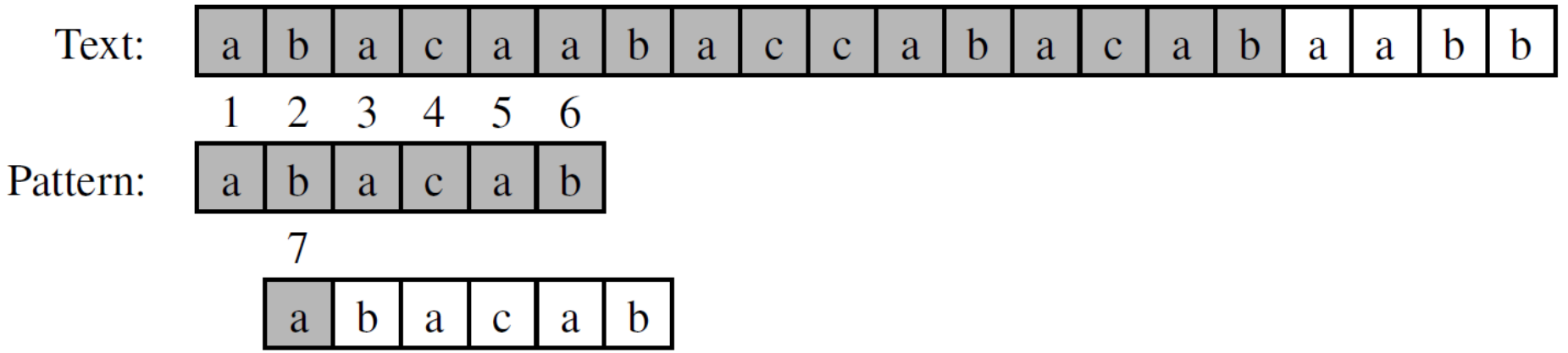
BRUTE FORCE PATTERN MATCHING

The basic pattern matching problem is to find whether a given string pattern (P) occurs in a larger body of text (P).



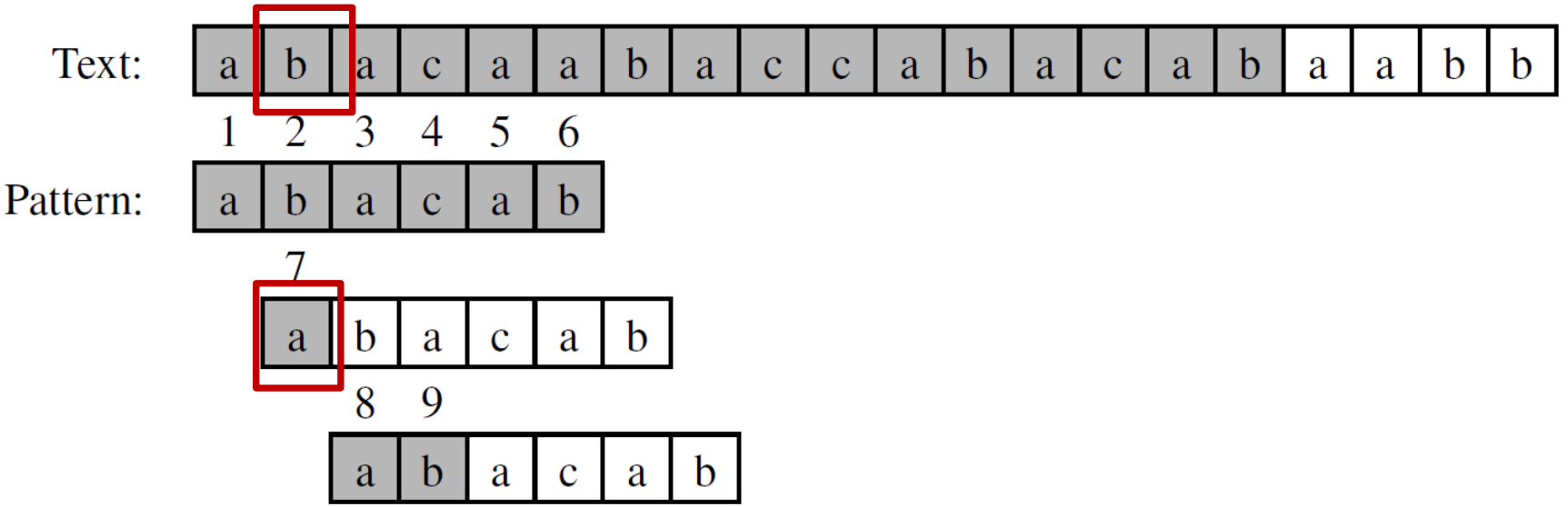
- In the above, characters 1 – 5 of the pattern **P** ('abaca') match those of the text **T**, but character 6 doesn't match the corresponding one in **T**

BRUTE FORCE PATTERN MATCHING



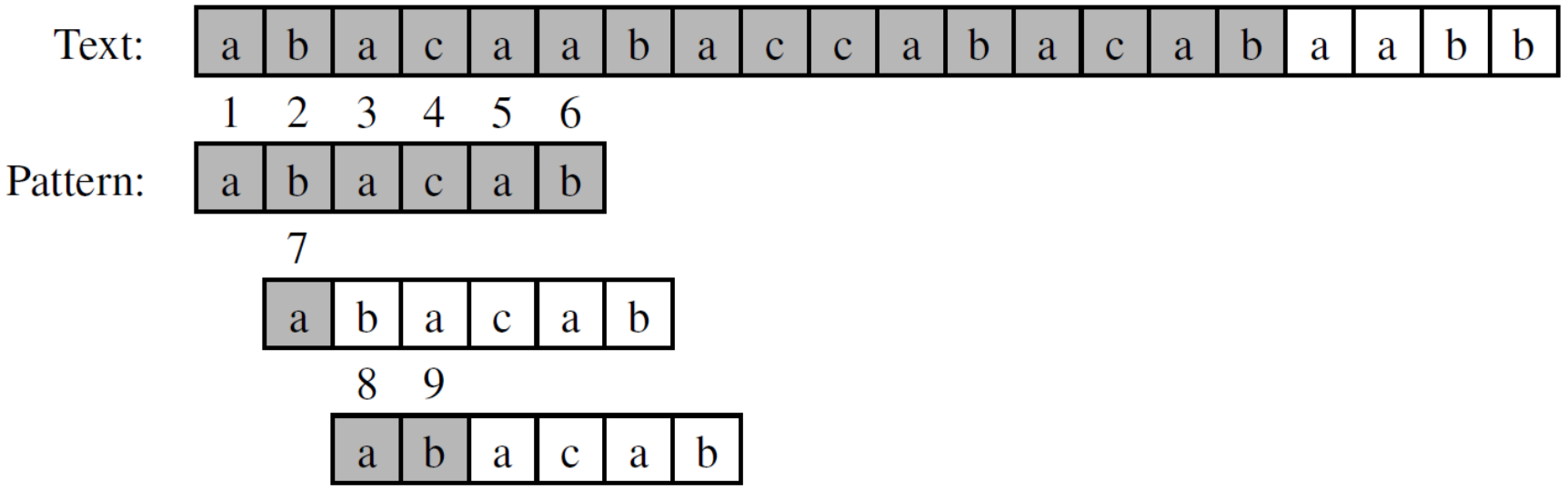
- So shift the pattern one character to the right relative to the text and repeat the character-by-character comparison

BRUTE FORCE PATTERN MATCHING



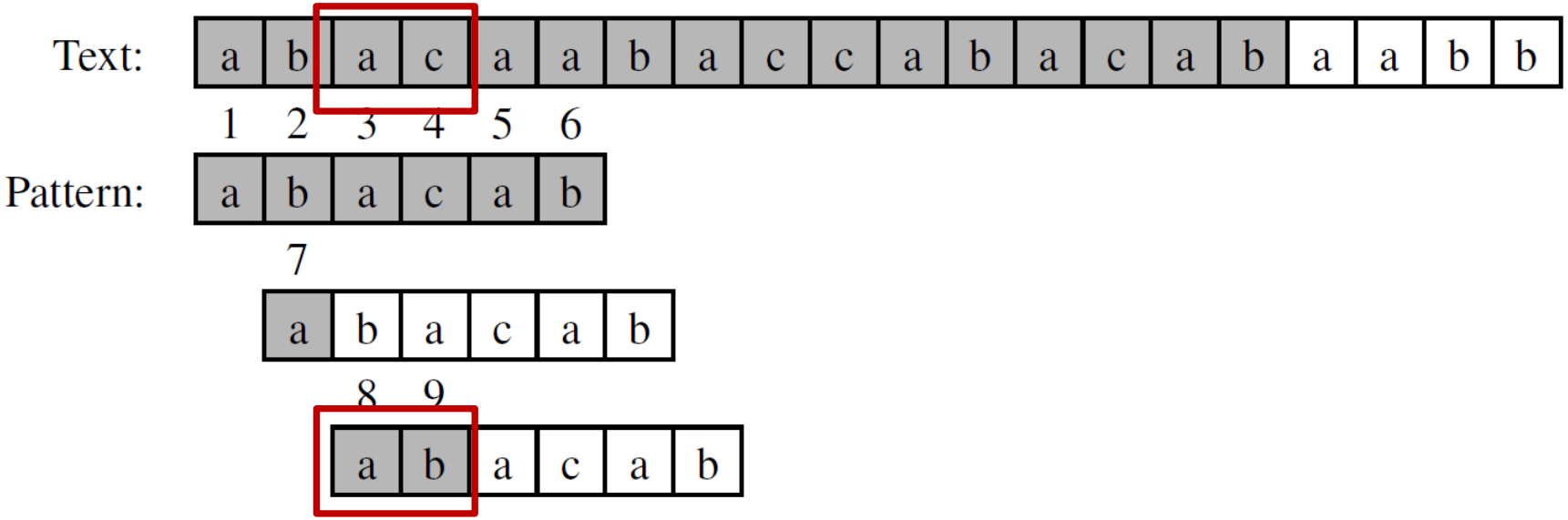
- In the above, the ‘a’ (7) doesn’t match the ‘b’ in the text, so…

BRUTE FORCE PATTERN MATCHING



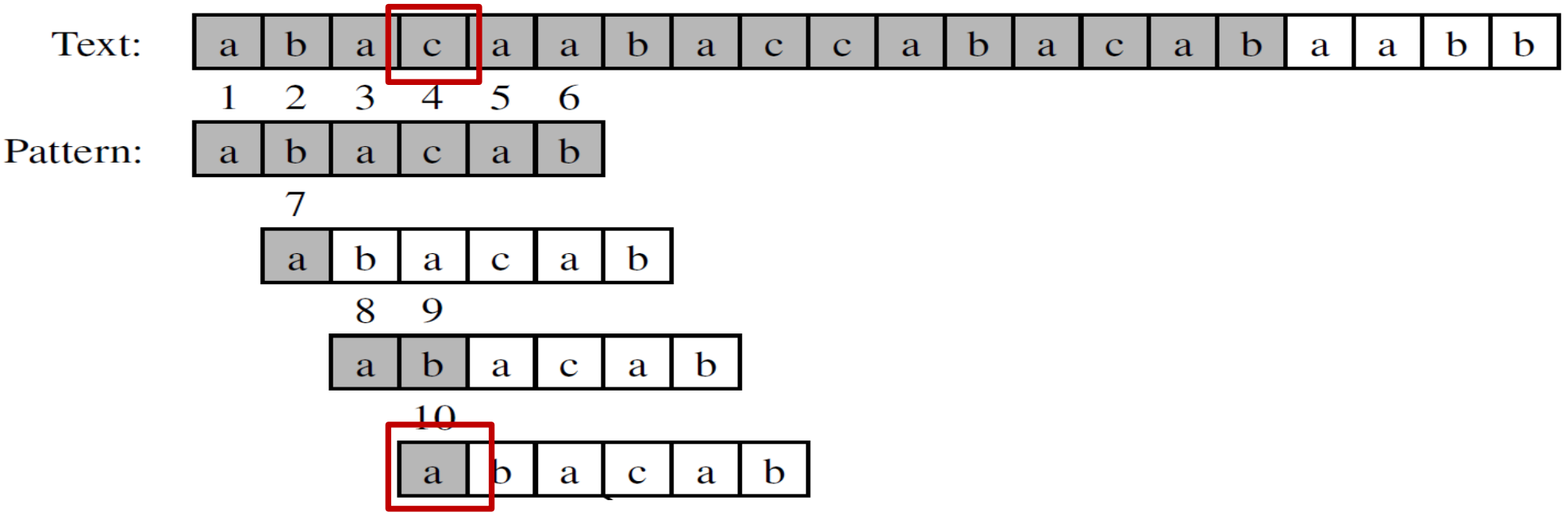
- … shift the pattern one character to the right and repeat the comparison

BRUTE FORCE PATTERN MATCHING



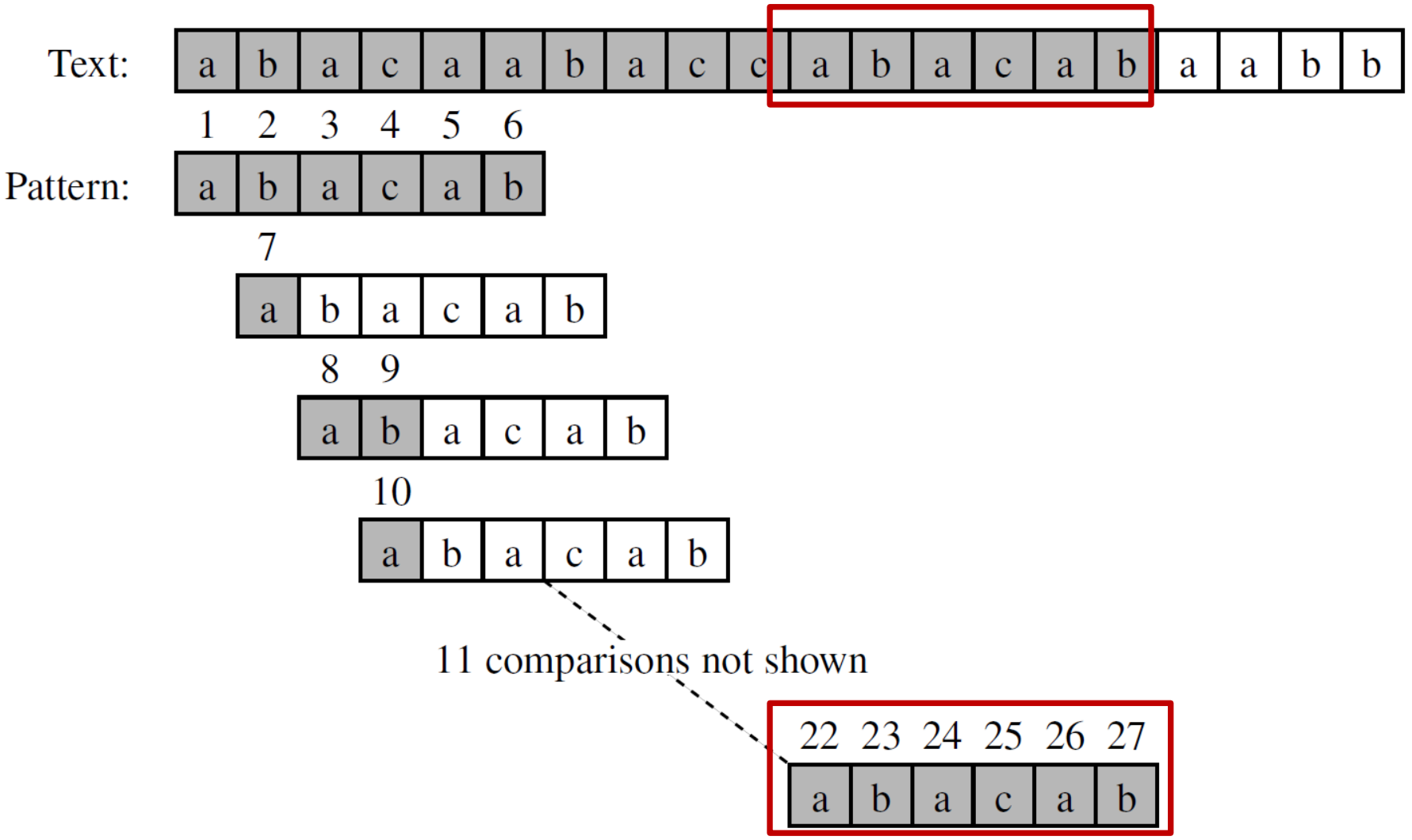
- in the above, the ‘a’ (8) matches the ‘a’, so continue checking
- but the ‘b’ (9) doesn’t match the ‘c’ in the text so shift after two comparisons (8 and 9)

BRUTE FORCE PATTERN MATCHING



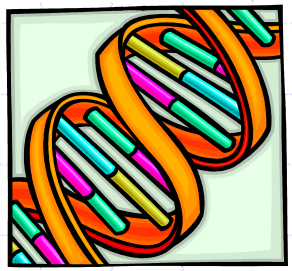
- in the above, the ‘a’ (10) doesn’t match the ‘c’ in the text, so shift

BRUTE FORCE PATTERN MATCHING



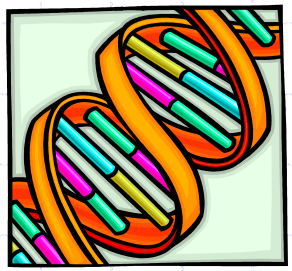
- continue the process for 11 more comparisons until a match is discovered with comparisons 22 - 27

Brute-Force Pseudo Code



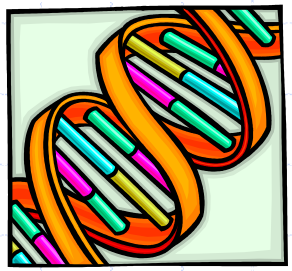
```
Algorithm BruteForceMatch(T, P)
    Input text T of size n and pattern
        P of size m
    Output starting index of a
        substring of T equal to P or −1
        if no such substring exists
    for i ← 0 to n − m
        { test shift i of the pattern }
        j ← 0
        while j < m ∧ 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 Pseudo Code



```
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}
```

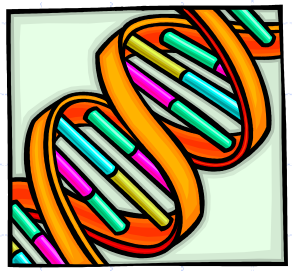
Brute-Force Pseudo Code



```
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}
```

Checks each character of P and T

Brute-Force Pseudo Code



```
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}
```

If j reached the size of P, that means all characters of the pattern matched a substring of T

PYTHON IMPLEMENTATION

```
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
```

Code Fragment 13.1: An implementation of brute-force pattern-matching algorithm.

BRUTE FORCE ALGORITHM PERFORMANCE

O(mn), where m = len(T) and n len(P)

Example:

- Text = 100 characters, pattern = 5 characters
- As the five character pattern “slides” across the 100 characters of text, they must be shifted at most 96 times (m – n + 1)
- At each shift at most five character comparisons must be done before shifting to the next text character
- 96 x 5 ≈ m x n, or ***O***(mn)

End of Text Processing – Brute Force

Please proceed to the next video now or at a later time