COMP1819
Algorithms and Data Structures

Lecture 11: Text Processing

Dr. Tuan Vuong

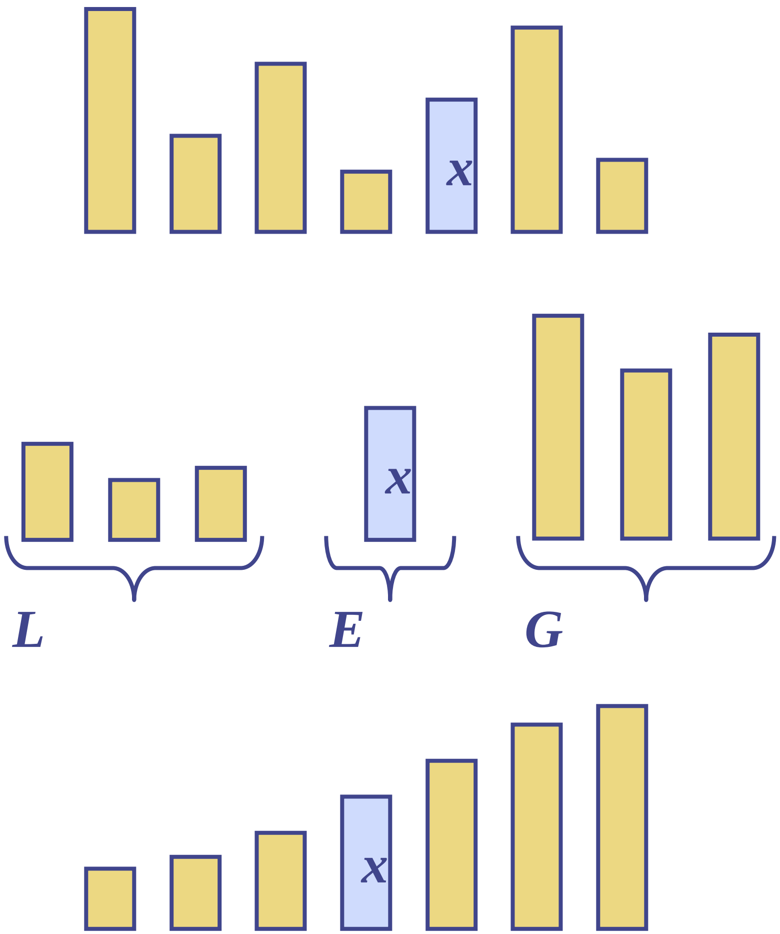
30/03/2021

Content

- Lab 10 Walk-through
- Text Processing/ Pattern-matching
- Assessments, Exam General info.
- Reinforcement

Quick-Sort

- Quick-sort is a randomized sorting algorithm based on the divide-and-conquer paradigm:
  - Divide: pick a random element x (called pivot) and partition S into
    - L elements less than x
    - E elements equal x
    - G elements greater than x
  - Recur: sort L and G
  - Conquer: join L, E and G



Review

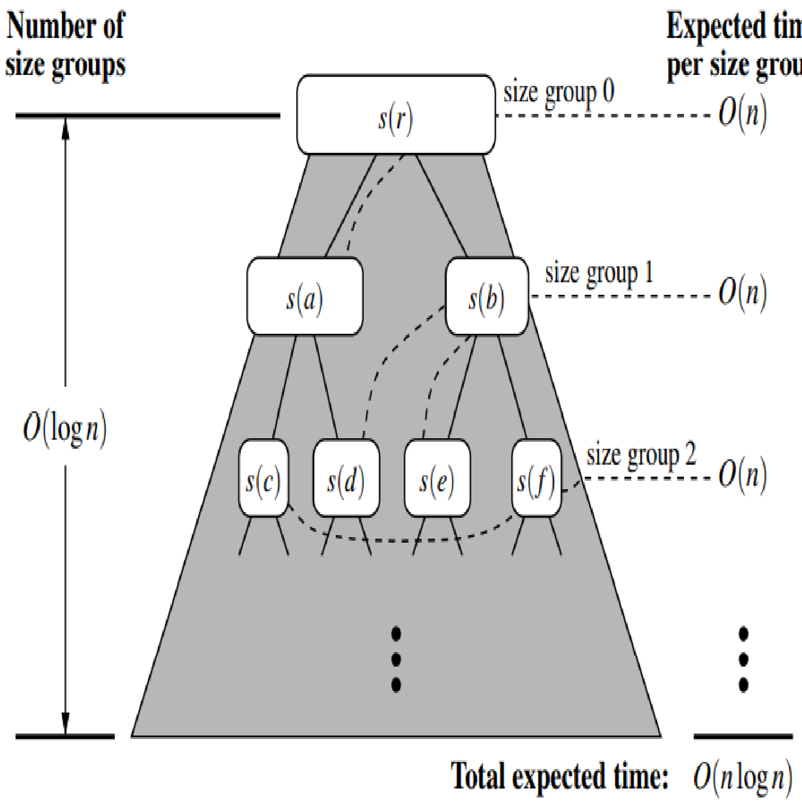


Figure 12.13: A visual time analysis of the quick-sort tree T. Each node is shown labeled with the size of its subproblem.

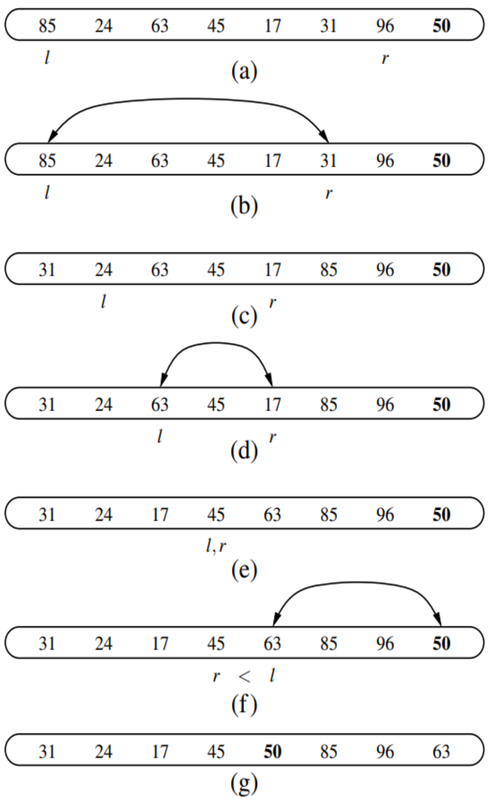


Figure 12.14: Divide step of in-place quick-sort, using index l as shorthand for identifier left, and index r as shorthand for identifier right. Index l scans the sequence from left to right, and index r scans the sequence from right to left. A swap is performed when l is at an element as large as the pivot and r is at an element as small as the pivot. A final swap with the pivot, in part (f), completes the divide step.

Understand quick sort and its complexity.

Walk-through

2. Sorts

Generate a random list of integers. Show how this list is sorted by the following algorithms:

- bubble sort
- selection sort
- insertion sort
- merge sort
- quick sort

Hints: if you implement all of the above, well done!

Understand the idea of merge sort and quick sort and its complexity.

Assessments

Coursework

- Programming assignment including a report
- Worth 50%

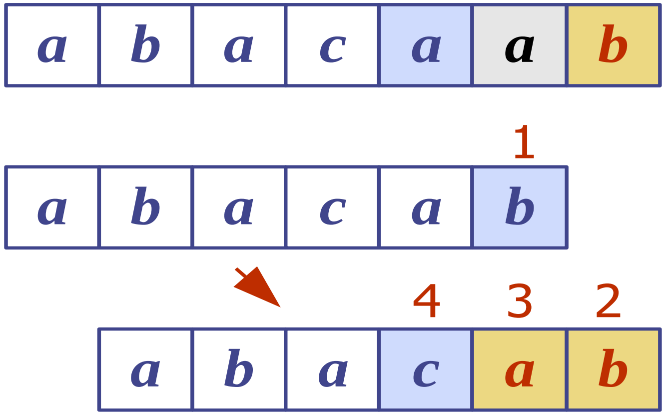
Exam

- Multiple choice, open book
- Worth 50%

Thursday 6/5/2021 9.30am Online

Please check your personal timetable regularly

Pattern Matching



Watch: https://www.youtube.com/watch?v=nK7SLhXcqRo

Abundance of digitised text

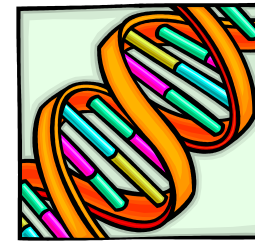
Internet documents, local documents, email, customer review, social status

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 $i \leftarrow 0$ **to** $n - m$
        { test shift i of the pattern }
        $j \leftarrow 0$
        while $j < m \wedge T[i + j] = P[j]$
            $j \leftarrow j + 1$
        if $j = m$
            return i {match at i}
        else
            break while loop {mismatch}
    return -1 {no match anywhere}

Boyer-Moore Heuristics

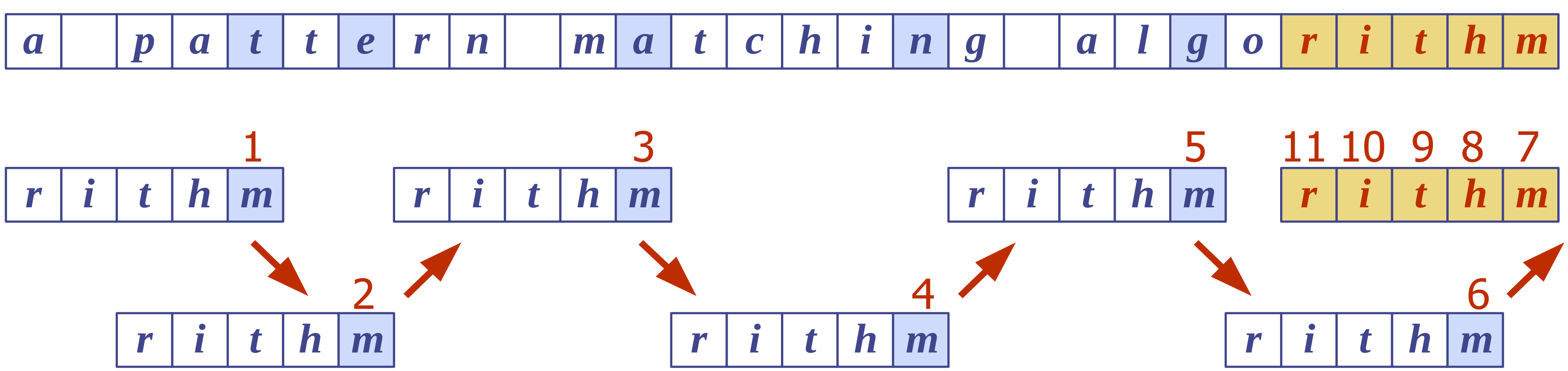
- The Boyer-Moore's pattern matching algorithm is based on two heuristics

  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

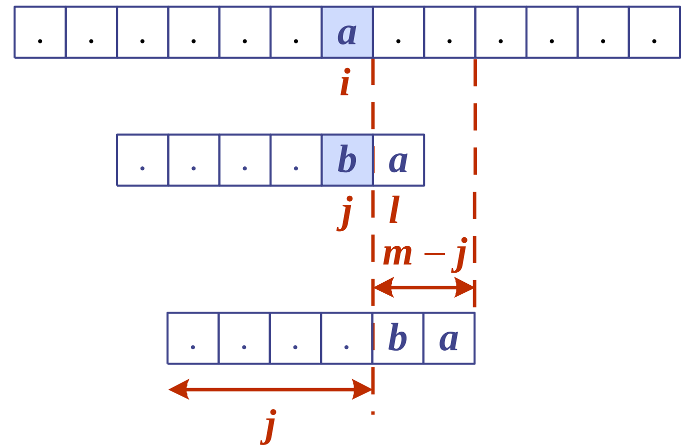
- The last-occurrence function can be represented by an array indexed by the numeric codes of the characters
- The last-occurrence function can be computed in time $O(m + s)$, where m is the size of P and s is the size of Σ

The Boyer-Moore Algorithm

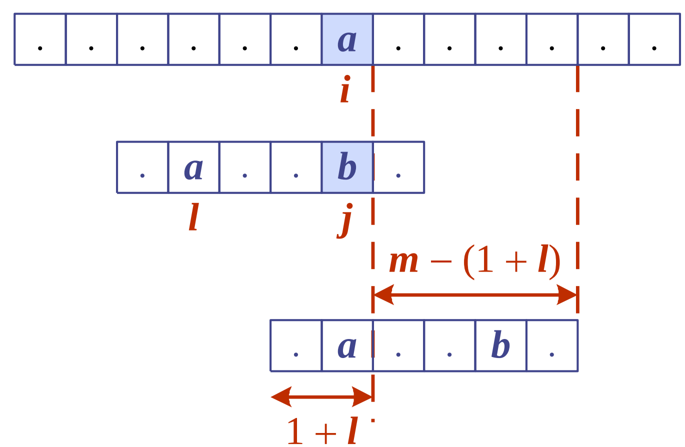
Algorithm *BoyerMooreMatch*(T, P, Σ)

$L \leftarrow$ *lastOccurenceFunction*(P, Σ)
$i \leftarrow m - 1$
$j \leftarrow m - 1$
repeat
  if $T[i] = P[j]$
    if $j = 0$
      return i { match at i }
    else
      $i \leftarrow i - 1$
      $j \leftarrow j - 1$
  else
    { character-jump }
    $l \leftarrow L[T[i]]$
    $i \leftarrow i + m - \min(j, 1 + l)$
    $j \leftarrow m - 1$
until $i > n - 1$
return -1 { no match }

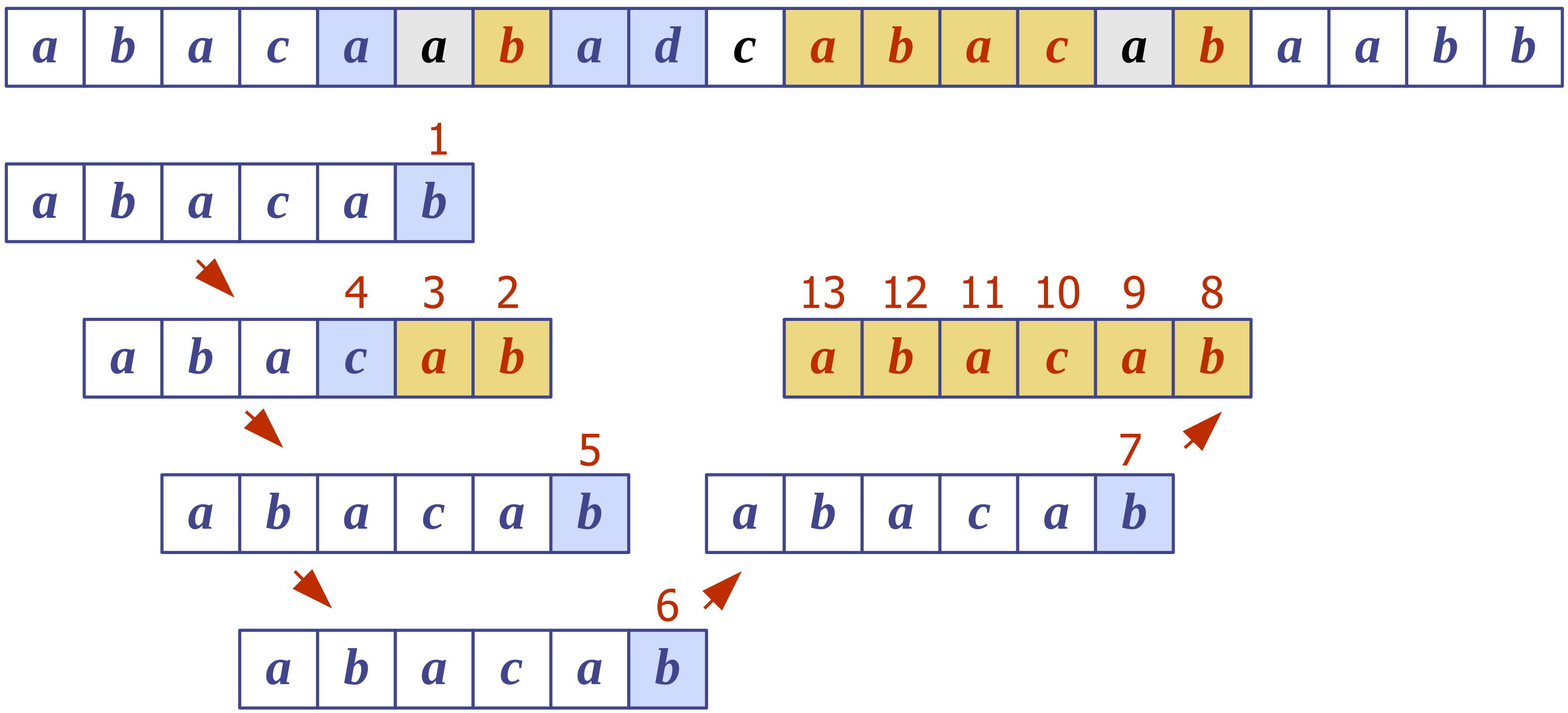
Case 1: $j \leq 1 + l$



Case 2: $1 + l \leq j$

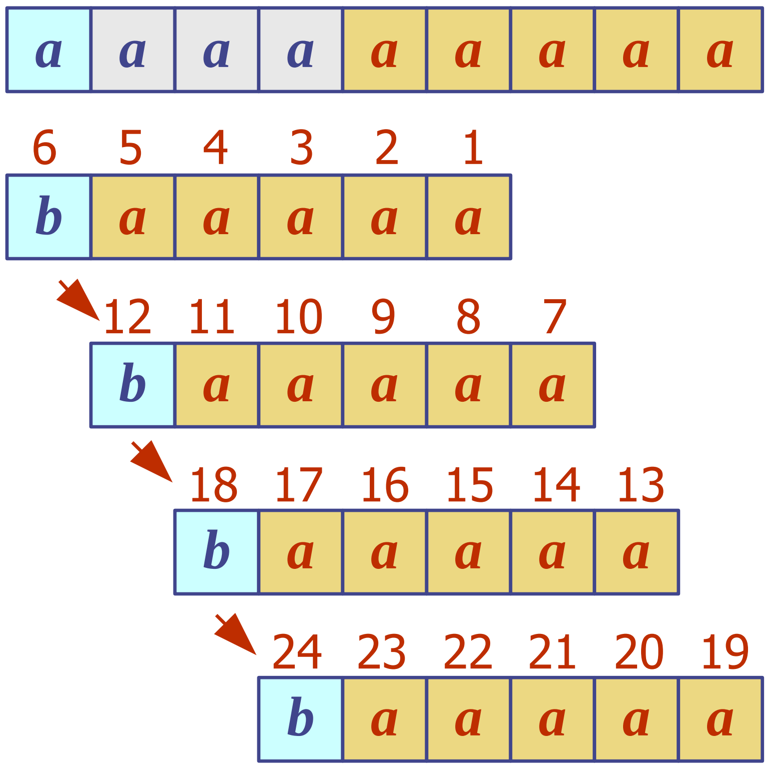


Example


a b a c a a b a d c a b a c a b a a b b
1
a b a c a b
4 3 2
a b a c a b
13 12 11 10 9 8
a b a c a b
5
a b a c a b
7
a b a c a b
6
a b a c a b

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

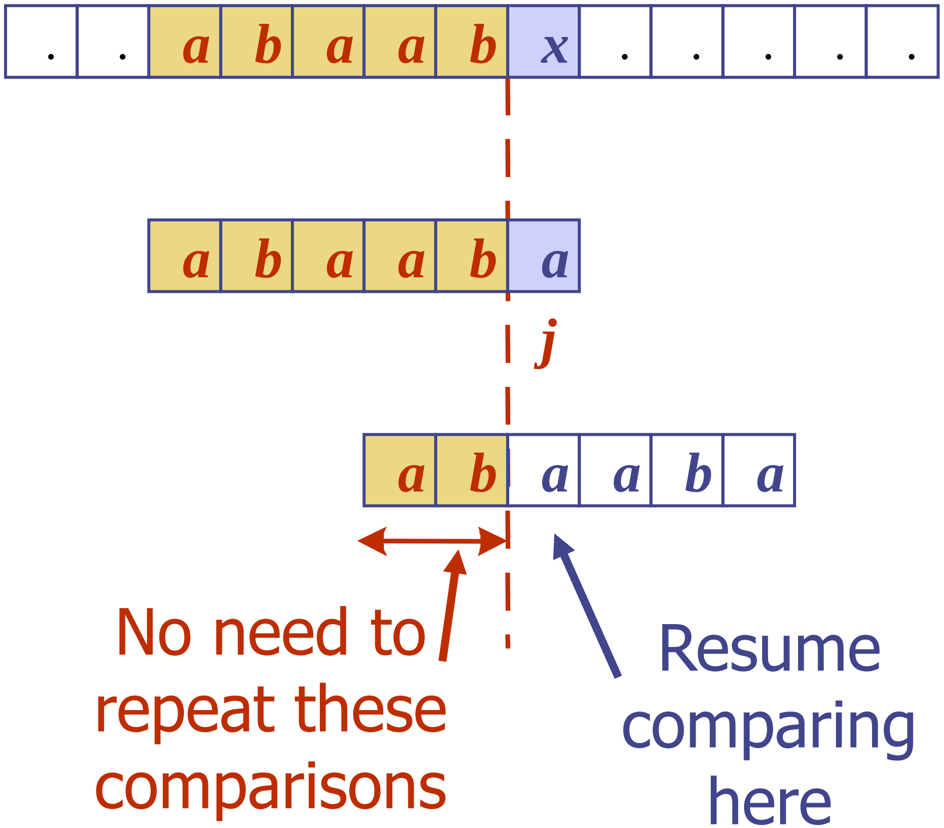


Python Implementation

```
def find_boyer_moore(T, P):
  """Return the lowest index of T at which substring P begins (or else -1)."""
  n, m = len(T), len(P)                    # introduce convenient notations
  if m == 0: return 0                      # trivial search for empty string
  last = { }                               # build 'last' dictionary
  for k in range(m):
    last[ P[k] ] = k                       # later occurrence overwrites
  # align end of pattern at index m-1 of text
  i = m-1                                  # an index into T
  k = m-1                                  # an index into P
  while i < n:
    if T[i] == P[k]:                       # a matching character
      if k == 0:
        return i                           # pattern begins at index i of text
      else:
        i -= 1                             # examine previous character
        k -= 1                             # of both T and P
    else:
      j = last.get(T[i], -1)               # last(T[i]) is -1 if not found
      i += m - min(k, j + 1)               # case analysis for jump step
      k = m - 1                            # restart at end of pattern
  return -1
```

The KMP Algorithm

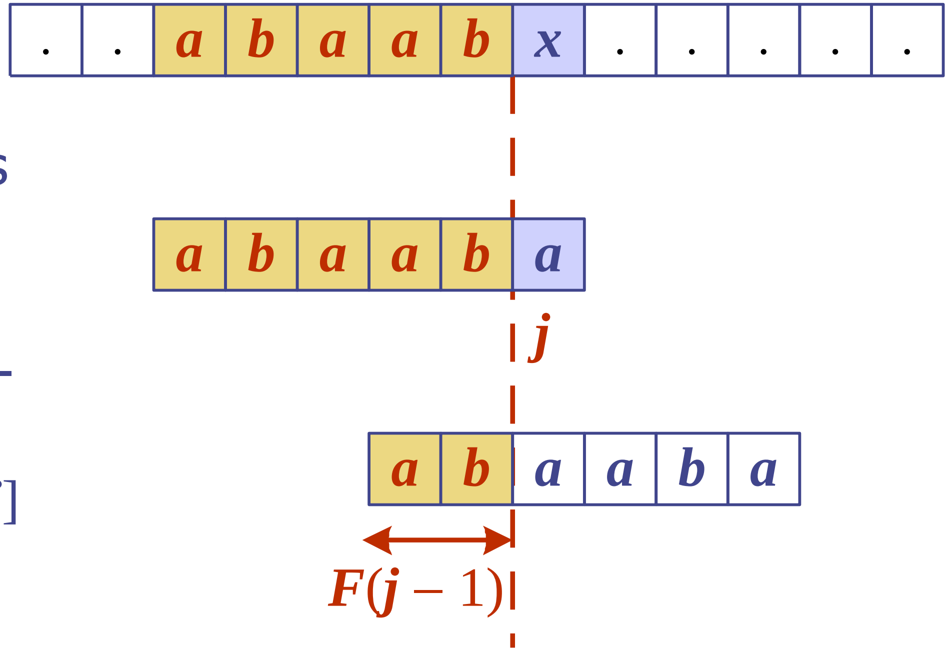
- Knuth-Morris-Pratt's algorithm compares the pattern to the text in left-to-right, but shifts the pattern more intelligently than the brute-force algorithm.
- When a mismatch occurs, what is the **most** we can shift the pattern so as to avoid redundant comparisons?
- Answer: the largest prefix of $P[0..j]$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

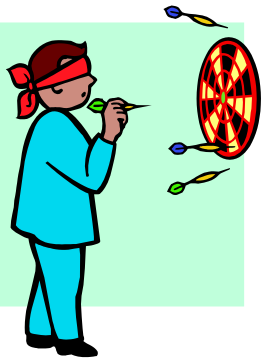


The KMP Algorithm

- 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

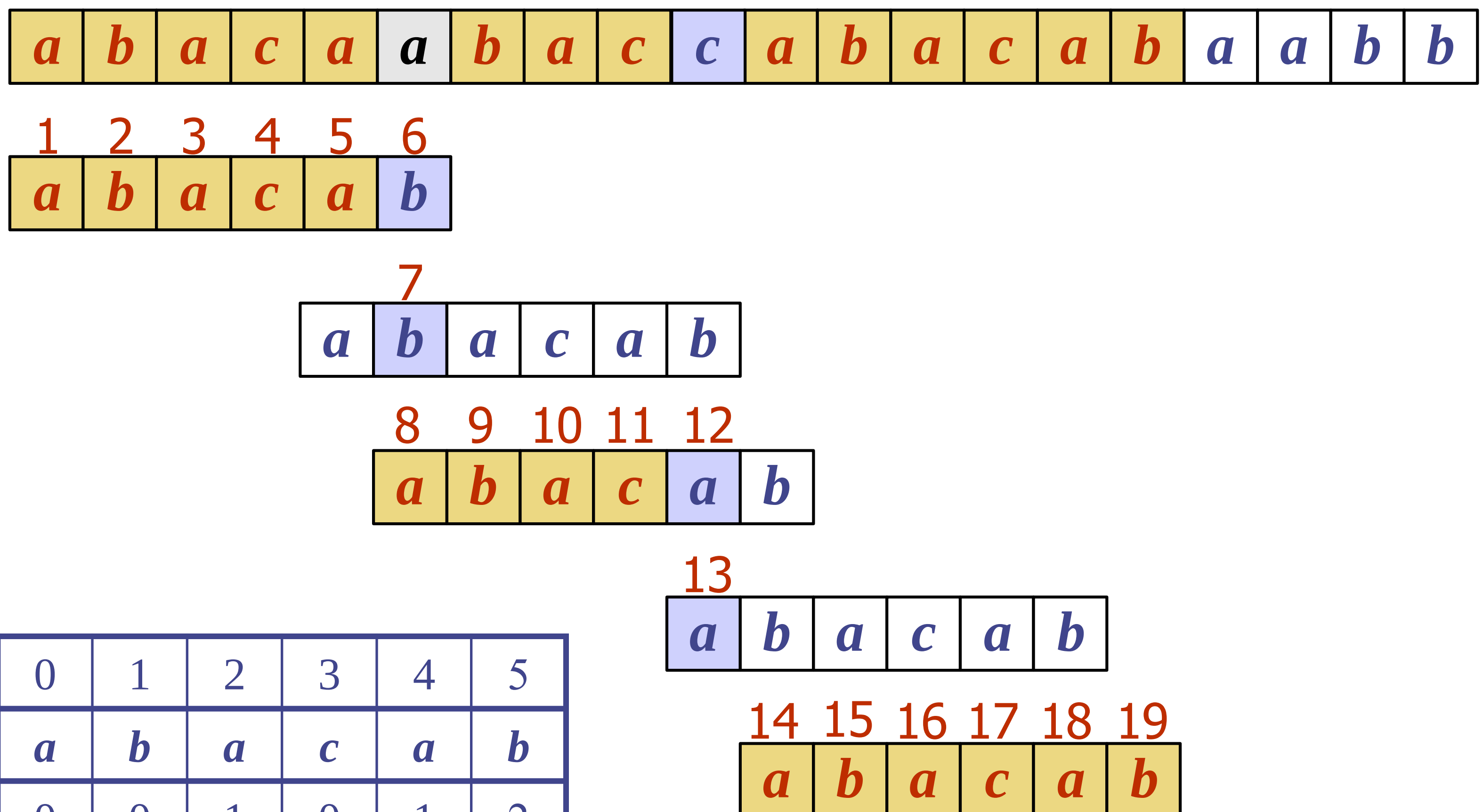


- The failure function can be represented by an array and can be computed in $O(m)$ time
- The construction is similar to the KMP algorithm itself
- At each iteration of the while-loop, either
  - i increases by one, or
  - the shift amount $i - j$ increases by at least one (observe that $F(j - 1) < j$)
- 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

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

```
def compute_kmp_fail(P):
  """Utility that computes and returns KMP 'fail' list."""
  m = len(P)
  fail = [0] * m                 # by default, presume overlap of 0 everywhere
  j = 1
  k = 0
  while j < m:                   # compute f(j) during this pass, if nonzero
    if P[j] == P[k]:             # k + 1 characters match thus far
      fail[j] = k + 1
      j += 1
      k += 1
    elif k > 0:                  # k follows a matching prefix
      k = fail[k−1]
    else:                        # no match found starting at j
      j += 1
  return fail
```

Reinforcement

Question

Draw a figure illustrating the comparison done by brute-force pattern matching for the text “aaabaadaabaaa” and pattern “aabaaa”

Question

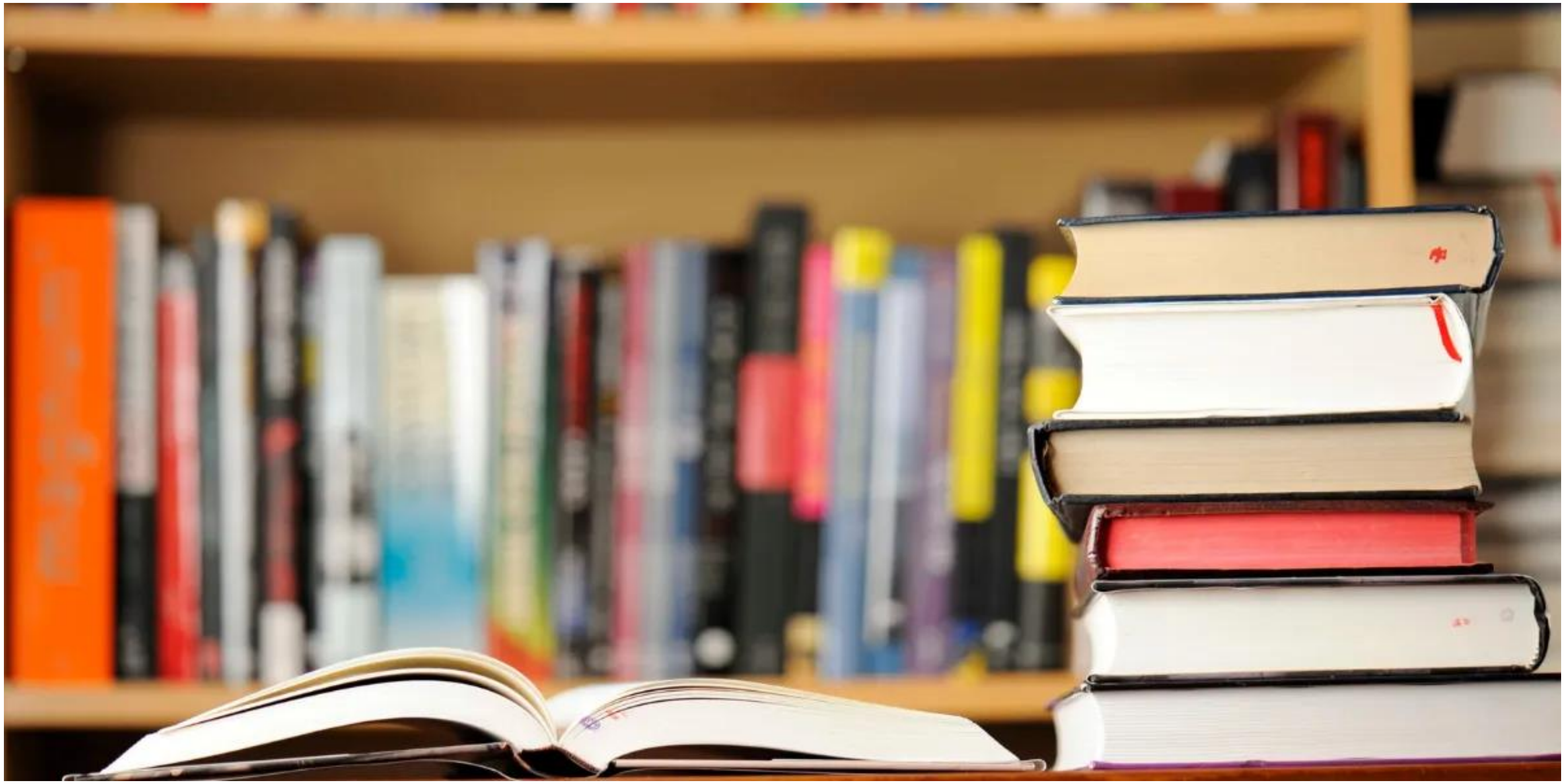
Design a worst case scenario by brute-force pattern matching for the text length 15 and pattern “aabaaa”

Quick overview

- Text-based pattern-matching is a very common problem in many application areas.
- Simple pattern-matching can be inefficient O (m*n)
- Better approaches: The Boyer-Moore Algorithm, Knuth-Morris-Pratt (KMP)

Extra reading

- Greed method: Krunack
- Dynamic Programming



Exam revision

Next week!