

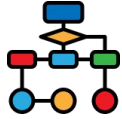
String Matching

Naive Algorithm
Rabin-Karp Algorithm
Knuth-Morris-Pratt Algorithm
Boyer-Moore Algorithm

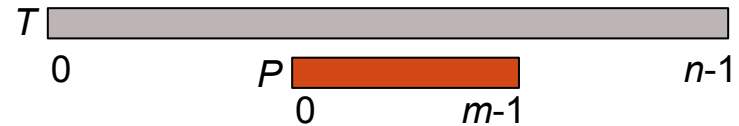
Ben Adams (slides by Richard Lobb & R. Mukundan)

benjamin.adams@canterbury.ac.nz

Department of Computer Science and Software Engineering
University of Canterbury



The String Matching problem



Text string containing n characters: $T [0 .. n-1]$,

Search pattern containing m characters: $P [0 .. m-1]$, $m < n$

- Determine whether P occurs in T
- Report every occurrence of P in T
- Generally, n is very large compared to m .

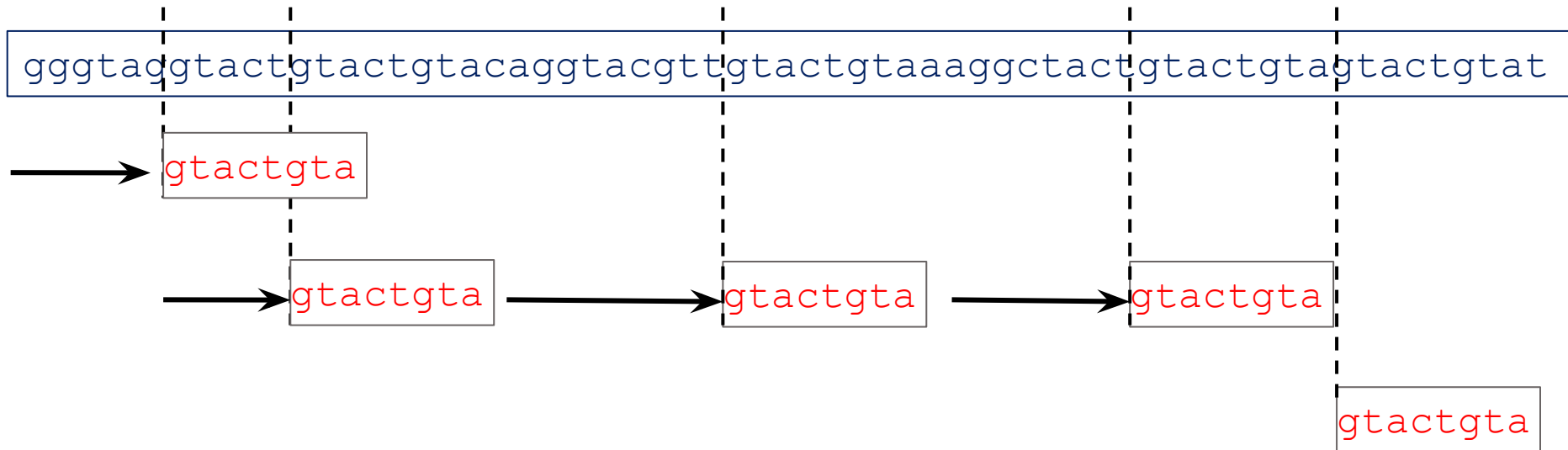
Applications:

- Search and replace functions in text editors
- Keyword searches in databases
- Internet applications
- DNA pattern matching

Example: DNA pattern matching

Human DNA: ~ 3 billion base-pairs (chars)

One gene: typically 29,000 base-pairs



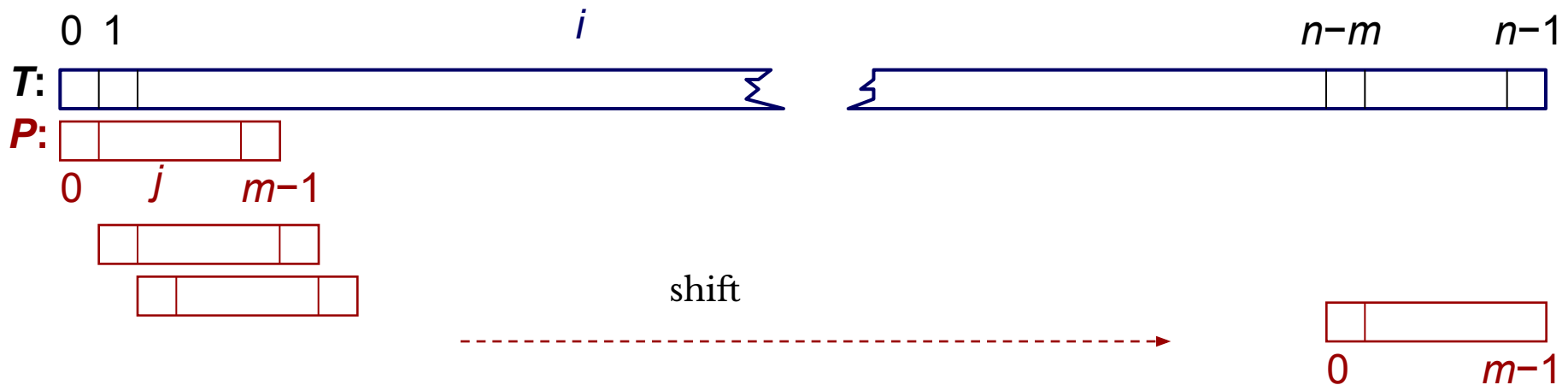
DNA Bases

DNA strings are drawn from the alphabet $\Sigma = \{a, c, g, t\}$

Naïve Algorithm

Shift the pattern from left to right, one character position at a time, and report all matching positions i such that

$$T[\text{shift}+j] == P[j], \text{ for } j = 0, 1, \dots, m-1.$$



```

for shift in range(0, n - m + 1):
    j = 0
    while j < m and T[shift+j] == P[j]:
        j = j+1
    if j == m:
        print(shift)
  
```

Naïve Algorithm

Watch it in action:

<https://lobb.nz/stringmatchvisualiser/stringmatching.html>

the mathematical theory
thema

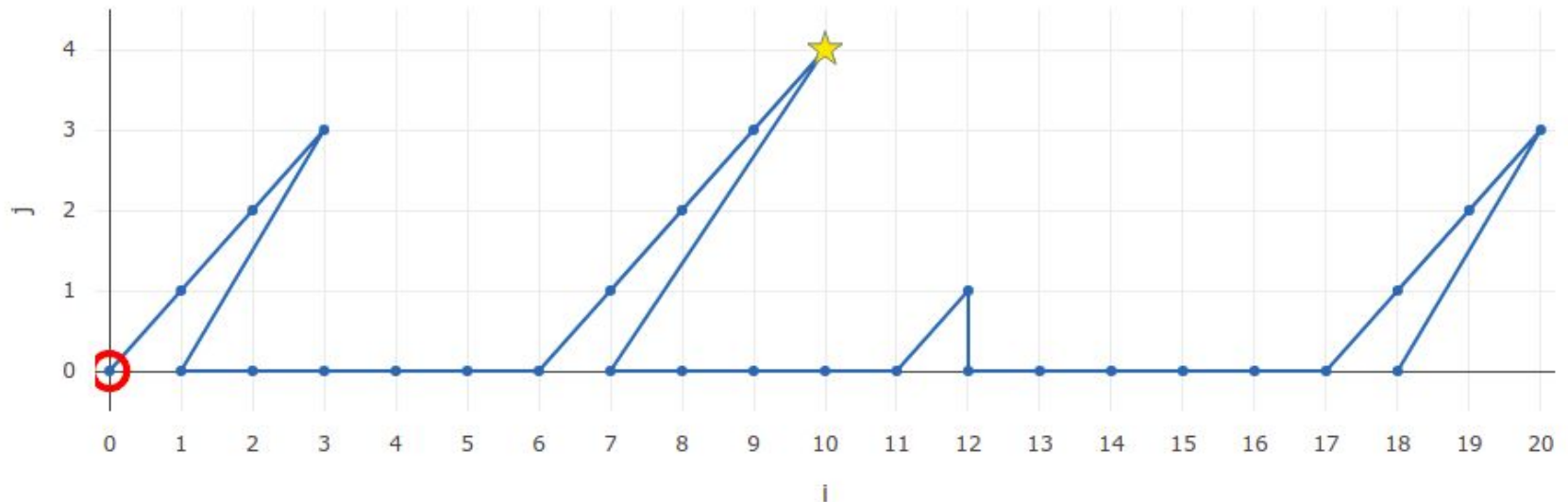
Next

Previous



Show state table

Comparison Trajectory



Naïve Algorithm - Complexity

Best case: $O(n)$

aaa
bbbbbbbbbb

Worst case: $O(nm)$

aab
aaaaaab

Expected: $O(n)$

With random alphabetic text, probability of mismatch in the first character is 25/26.

A good (enough) algorithm with typical written text and shortish patterns.


Python String Matching

ASIDE

Built-in pattern matching (optimised string search) functions:

str.find() (exact matching - naïve algorithm??)

```
text = 'Python string matching algorithms'
pattern = 'ing'
index = text.find(pattern)
while index > -1 :
    print (index)
    index = text.find(pattern, index+1)
```

10, 19

Sequence Matcher (partial matching - Ratcliff-Obershelp algorithm)

```
from difflib import SequenceMatcher
text = 'ctgacttggcactcg'
pattern = 'tgactacgcac'
s = SequenceMatcher(None, text, pattern)
lyst = s.get_matching_blocks()
print(lyst)
```



```
[Match(a=1, b=0, size=5), Match(a=8, b=7, size=4), Match(a=15, b=11, size=0)]
```


Improved Pattern Matching

- The naïve algorithm does not remember the structure of previously matched strings.
- The search pattern needs to be first preprocessed in order to analyze its structure.
 - **Pattern Preprocessing Algorithm**
- We consider three important algorithms for exact pattern matching:
 - **Rabin-Karp (RK) Algorithm**
 - **Knuth-Morris-Pratt (KMP) Algorithm**
 - **Boyer-Moore (BM) Algorithm**

Matching by hashing

Consider the following code to look for a pattern in a string s :

```
s = "asdfa sdfsfadsf areadgsasdasdaesasdfadeq"  
pat = "asdfa"  
m, n = len(pat), len(s)  
hash_pat = hash(pat)  
for i in range(n - m + 1):  
    if hash(s[i : i + m]) == hash_pat:  
        print(f"Possible match at position {i}")
```

which prints

```
Possible match at position 4  
Possible match at position 34
```

But $hash(s[i : i + m])$ is $O(m)$, so this code is $O(nm)$ even without hash collisions. Much worse than naive algorithm.

Rabin-Karp algorithm

Precompute a hash of the pattern, $hash-P$ $O(m)$

Think of the pattern as a sliding window over the text

for each possible window position: $O(n)$

 Compute hash of text in window, $hash-W$ by updating hash from previous position

$O(1)$



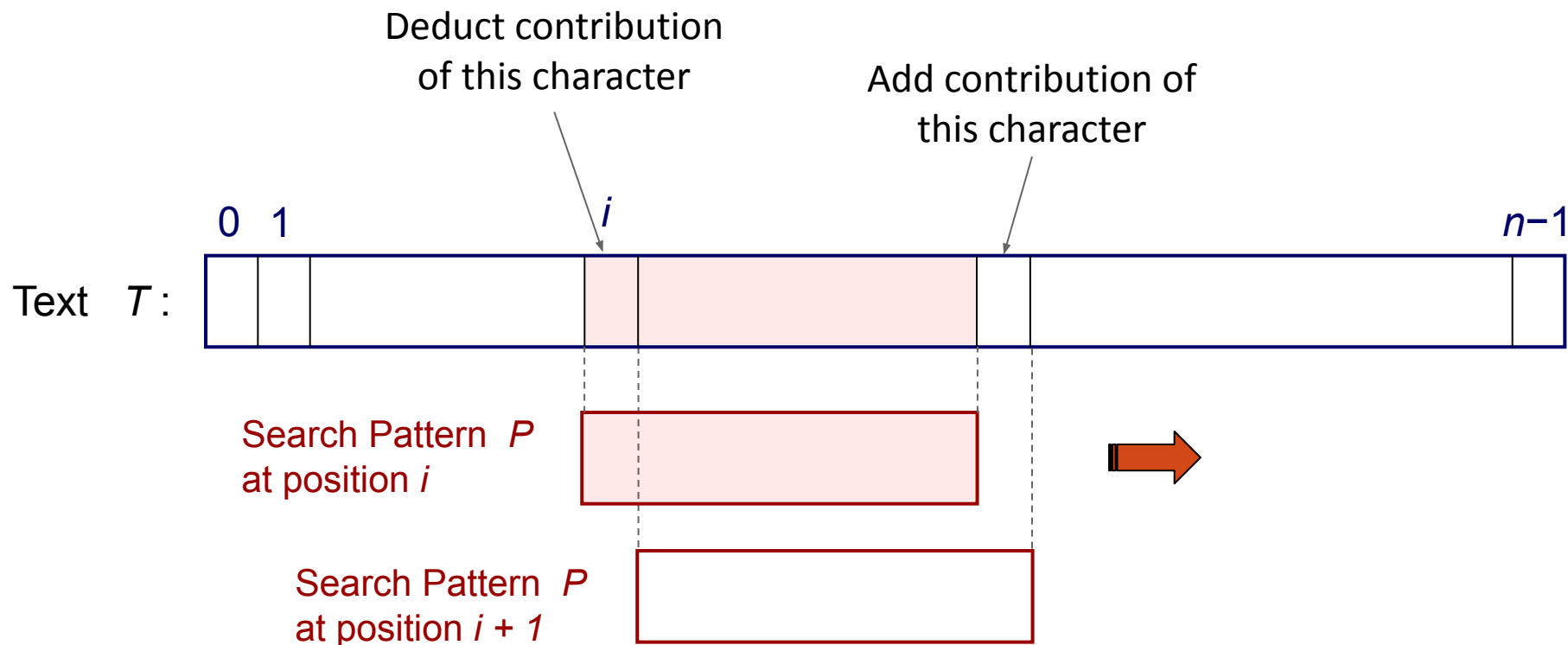
if $hash-P == hash-W$

if windowed text == pattern
 print matching position

Trick is to find a hash function with this property.

- Expected performance: $O(n + m)$
- Worst case: $O(mn)$ - checks all chars for every hash hit
- Good if multiple patterns P being matched (e.g. *turnitin*??)

Rabin Karp: hash update



Rabin-Karp Hashing Function

A "rolling hash"

Define

$$\text{hash}(t_i \dots t_{i+m-1}) = (t_i 2^{m-1} + t_{i+1} 2^{m-2} + \dots + t_{i+m-1} 2^0) \bmod q$$

where $t_i \dots t_{i+m-1}$ is the text in the current window and q is a large prime number.

Then to update the hash for the window $t_{i+1} \dots t_{i+m}$ we have

$$\begin{aligned} \text{hash}(t_{i+1} \dots t_{i+m}) &= (2 \times \text{hash}(t_i \dots t_{i+m-1}) - t_i 2^m + t_{i+m}) \bmod q \\ &= (2 \times \text{hash}(t_i \dots t_{i+m-1}) - t_i (2^m \bmod q) + t_{i+m}) \bmod q \end{aligned}$$

Written this way as 2^m could be massive.

Precompute this

Although straightforward, hash update is relatively expensive.

An improvement

Get far fewer hash collisions if use a bigger base, $b > |\text{alphabet}|$

e.g. $b = 256$

$$\text{hash}(t_i \dots t_{i+m-1}) = (t_i b^{m-1} + t_{i+1} b^{m-2} + \dots + t_{i+m-1} b^0) \bmod q$$

Update then becomes:

$$\begin{aligned} \text{hash}(t_{i+1} \dots t_{i+m}) &= (b \times \text{hash}(t_i \dots t_{i+m-1}) - t_i b^m + t_{i+m}) \bmod q \\ &= (b \times \text{hash}(t_i \dots t_{i+m-1}) - t_i (b^m \bmod q) + t_{i+m}) \bmod q \end{aligned}$$

Written this way as b^m could be massive

Precompute this

Python code

```

BASE = 256
Q = 15487469 # Any largish prime

def rabin_karp_search(pat, text):
    m, n = len(pat), len(text)
    i = j = 0
    hash_p = 0 # hash value for pattern
    hash_t = 0 # hash value for text
    h = 1      # For calculating BASE^m % Q

```

```

    for i in range(m):
        h = (h * BASE) % Q
        hash_p = (BASE * hash_p + ord(pat[i])) % Q
        hash_t = (BASE * hash_t + ord(text[i])) % Q

```

```

    for i in range(n - m + 1):
        if hash_p == hash_t:
            if pat == text[i : i + m]:
                print("Pattern found at index " + str(i))

```

```

    if i < n - m: # To prevent index error at end
        hash_t = (Q + BASE * hash_t - ord(text[i]) * h + ord(text[i + m])) % Q # Update hash

```

Searching complete works of Shakespeare for a single string gave typically 0 or 1 hash mis-hits. But naïve algorithm 3 x faster.

Compute BASE^m % Q
 # Computing pattern hash
 # Computing window hash for i = 0

 # For each window position

 # Char-by-char comparison only on a hash hit

When Rabin-Karp really wins ...

- ... is when you're searching for any/all of a large number of patterns.
- Build a *set* of the hashes of all search patterns
- At each step, check if window hash is in the set
 - $O(1)$
- Is this what *turnitin* does??

Improving naive algorithm on a mismatch

Consider

T: aaaaaabaaaaaabaabaaaaabaaaaabaaaaab
 P: aaaaaaa
 P': aaaaaaa

Mismatch

- Clearly, on a mismatch *in this case*, we can restart at P'

But with

T: aaaaaabaaaaaabaabaaaaabaaaaabaaaaab
 P: aaaaaab
 P': aaaaaab

Mismatch

- Now on a mismatch we can only move the pattern 1 step, but we **don't** need to go back to check P[0], P[1], ... P[m - 2]

DFA pattern recognisers

T: aaaaaabaaaaaabaabaaaaabaaaaabaaaaaab
 P: aaaaaaa
 P': aaaaaaa

Mismatch

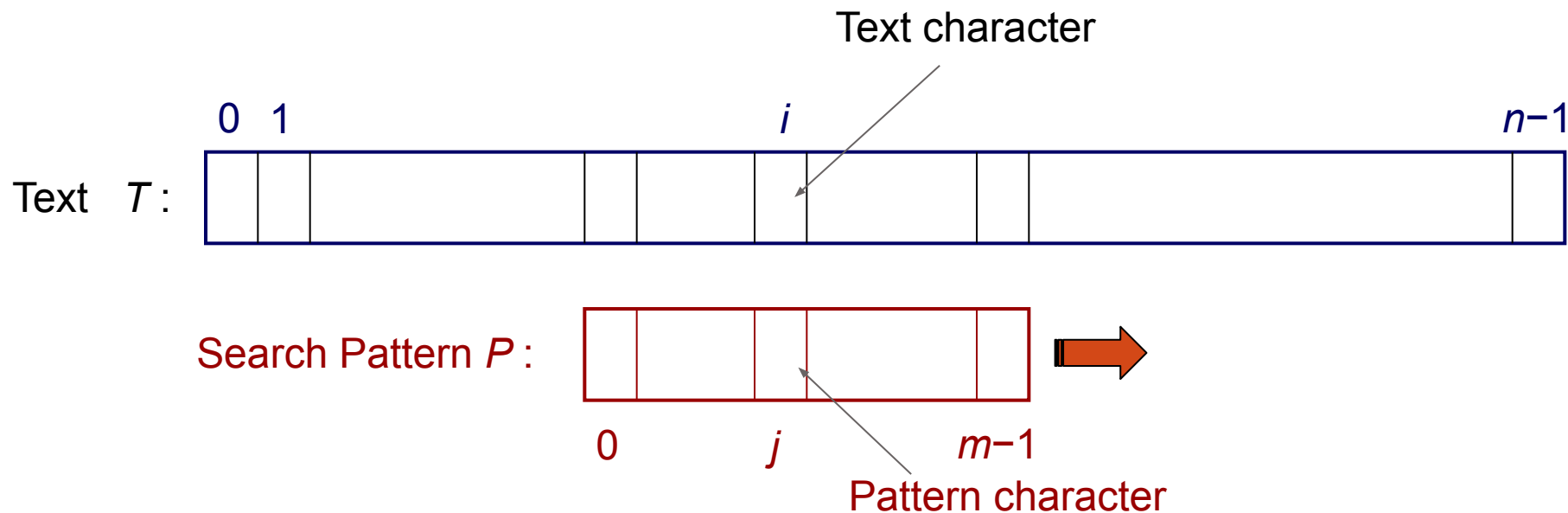
i.e. the next *shift* to try

- In general, we can precalculate the restart position for all possible mismatch positions j and characters $T[i]$
- Build a *Deterministic Finite State Automaton (DFA)*
 - See COSC261
- Checks $T[i]$ exactly once - $O(n)$
- But the DFA is usually too expensive to build (space and time)
- So we trade off slightly slower searching for faster preprocessing.

Knuth-Morris-Pratt (KMP) Algorithm

- Uses a pre-process to skip forwards after a failed match
 - Identifies repeated sequences in the pattern
 - Like a DFA, but with a fast approximate preprocess
- We *never* back up in the text, only in the pattern.
- Pre-process $O(m)$ in space and time
- Searching: $O(n)$ in time
 - At most $2n - 1$ character comparisons
- Overall: $O(n + m)$

String Matching Notation



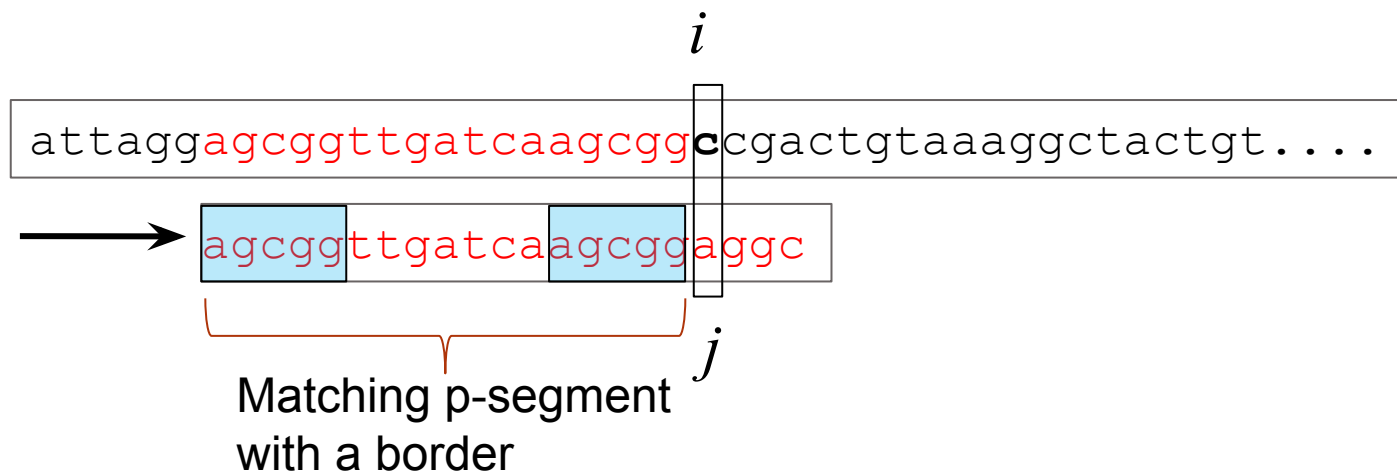
Note : Here i represents the current character position in the text.
 $P[j]$ is compared with $T[i]$.

Text: $T = t_0 t_1 \dots t_{n-1} = T[0:n]$
 Pattern: $P = p_0 p_1 \dots p_{m-1} = P[0:m]$

Matching Prefix Segment ("P-segment")

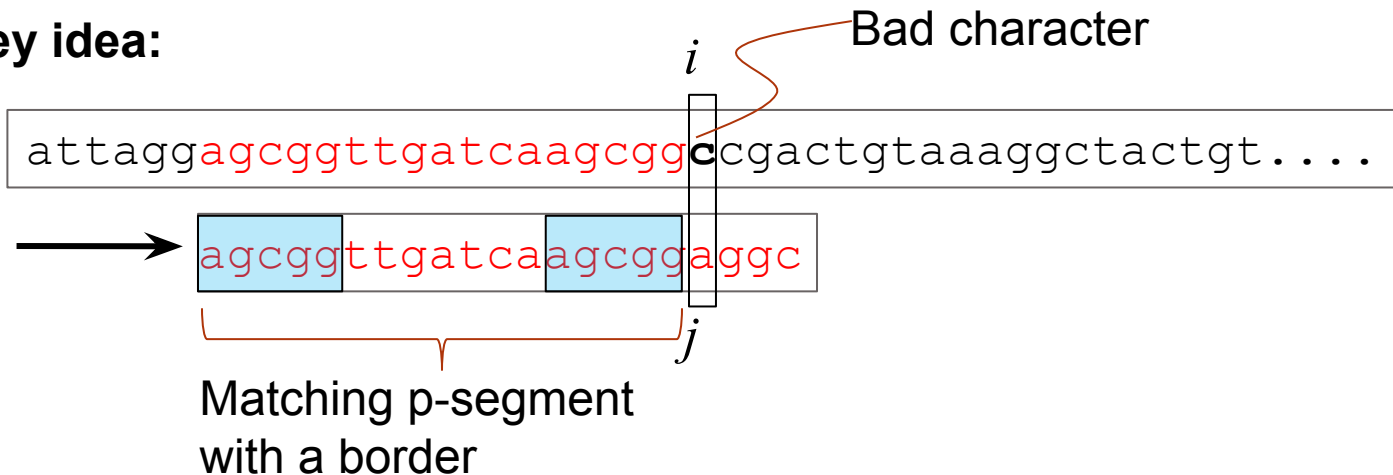
I introduce this term to reduce confusion with other prefixes in the algorithm.

- Suppose that the p-segment of the pattern $P[0:j]$ has matched with the corresponding portion of the text.
- Match failed at $P[j]$, i.e. $T[i] \neq P[j]$

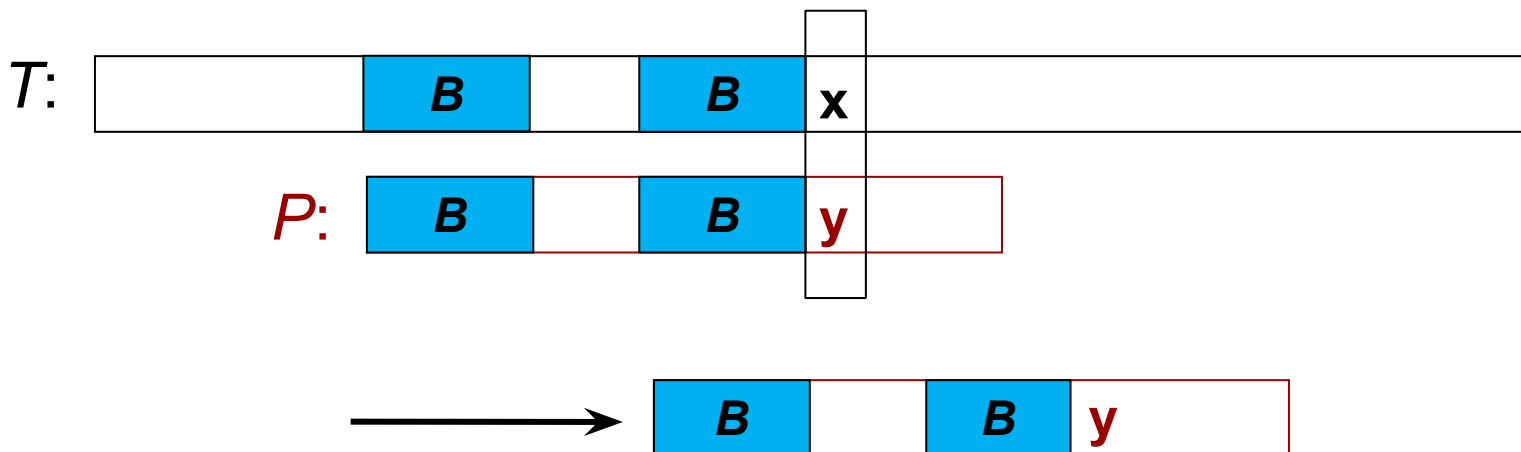


Matching P-segment and Bad Character

Key idea:



If B is the **widest** border of the matched p-segment of the pattern, the *minimum* shift worth trying is $j - \text{len}(B)$ to align B again. No border \Rightarrow shift by j .



P-segment Borders

- We can use the information about the border of a matching prefix to shift a pattern.
- We will develop a method to pre-compute the *widest* borders of all p-segments of a given pattern, and store the border lengths in an array $b[j]$:

j :	0	1	2	3	4	5	6	7	8	9	10	11
Pattern:	M	A	T	H	E	M	A	T	I	C	S	
$b[j]$:	-1	0	0	0	0	0	1	2	3	0	0	0

The length of the widest border of matching prefix MATHEMA

- We can use $b[j]$ to shift the pattern when a bad character in the text corresponding to location j is found.
- $b[0] = -1$ for use as a terminator; see later.

Calculating borders: a brute-force approach

The brute-force way to calculate the length of the maximum borders of s is to try all border lengths from 1 up to $\text{len}(s) - 1$.

Need to do this for all P-segments.

```
def border_lengths(pattern):  
    """A simple intuitive but inefficient implementation"""  
    b = [0] * (len(pattern) + 1)      # Border lengths  
    b[0] = -1                          # Acts as a terminator - see later  
    for i in range(1, len(pattern) + 1): # For each p-segment  
        p_seg = pattern[:i]  
        for j in range(1, i):  
            if p_seg[:j] == p_seg[-j:]: # Is it a border?  
                b[i] = j                # If so, save length  
    return b
```

But this is $O(m^3)$ - unacceptable. We need a better algorithm ...

Prefixes and Suffixes

Let $P = p_0 p_1 \dots p_{m-1}$ be a string of length m .

Let $p_{-1} = p_m = \varepsilon$ (null string of length 0).

A **proper prefix of P** is a substring S of P such that

$$S = p_0 \dots p_{k-1}, \quad (\text{or } P[:k]), \quad 0 \leq k < m.$$

A **proper suffix of P** is a substring V of P such that

$$V = p_{m-k} \dots p_{m-1}, \quad (\text{or } P[-k:]), \quad 0 < k < m.$$

Example:

Proper prefixes: $\varepsilon, g, gc, gct, gcta, gctag$

Proper suffixes: $\varepsilon, c, gc, agc, tagc, ctagc$

-1	0	1	2	3	4	5	6
ε	g	c	t	a	g	c	ε

$m = 6$

Border of a String

A string that is both a proper prefix and a proper suffix of P is called a **border of P** .

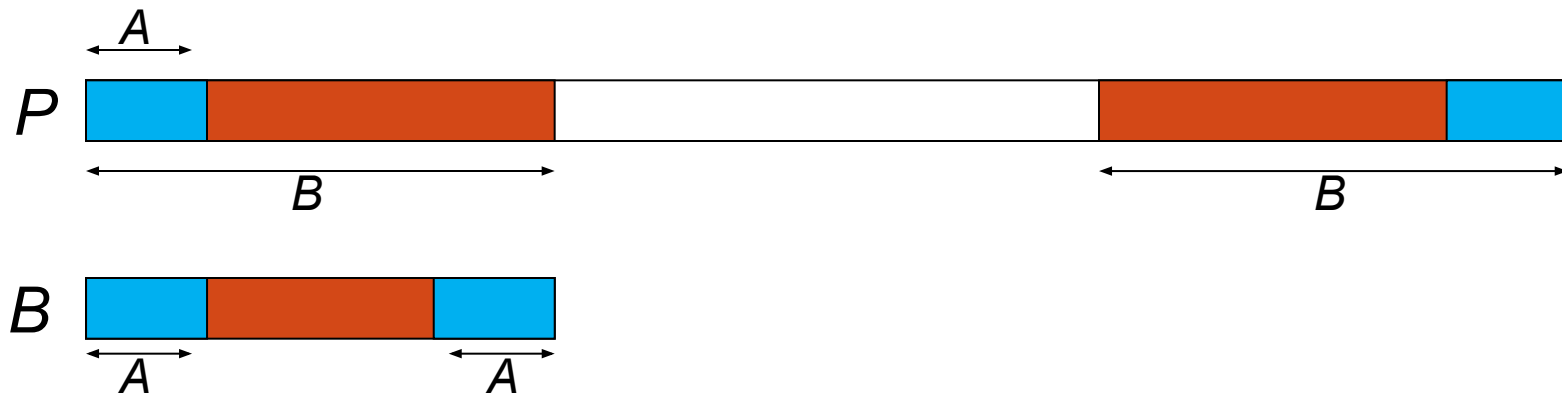
Example: $P = \text{gctgcacggctgc}$

Borders of P : ϵ , gc, gctgc

(ϵ is a border of every string).

If A , B are both borders of P , and if $|A| < |B|$, then A is also a border of B .

(In the above example, 'gc' is a border of 'gctgc')



Nested Borders

In general, if a string has several borders, they will always be nested within each other in a hierarchical structure.



Example:

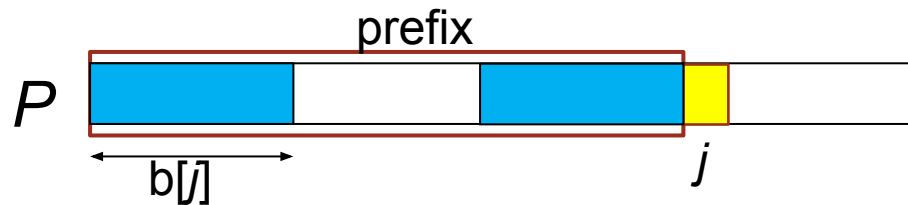
tagtacgtgtagtagcttagaggcttagtacgtgtagta

From the definition of a proper prefix/suffix, a string cannot be a border of itself.

So a single character cannot have a border other than ϵ .

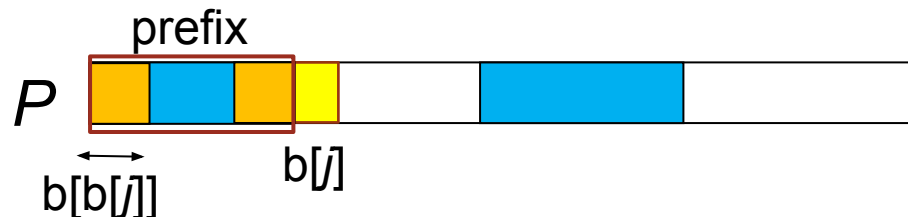
Fast Computation of P-segment Borders $b[j]$

Note: Here, by 'p-segment' at j , we mean the portion of the text $p_0 \dots p_{j-1}$. ($P[0:j]$)



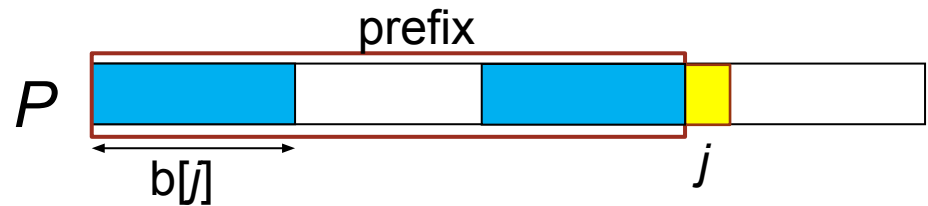
$b[j]$ represents the length of the widest border of the p-segment at position j .

$b[b[j]]$ represents the length of the widest border nested within the previous border of length $b[j]$, and so on...



Computation of P-segment Borders $b[j]$

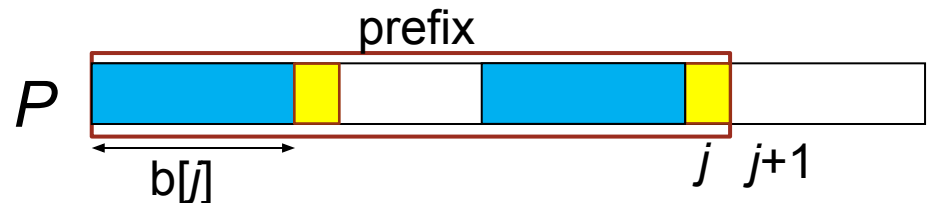
- Initialization: $b[0] = -1$; $b[1] = 0$.
- Sequential computation:



Can we compute $b[j+1]$, given $b[j]$?

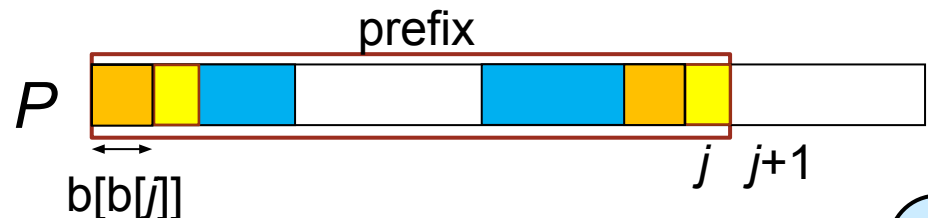
- If $P[j] == P[b[j]]$, then the border can be extended by one character:

$$b[j+1] = b[j] + 1$$



- Else, if $P[j] == P[b[b[j]]]$, the nested border can be extended by one character:

$$b[j+1] = b[b[j]] + 1$$



- And so on

Preprocessing Patterns: Example ($m = 1$)

j :	0	1	2	3	4	5	6	7	8	9	
<i>Pattern</i> :	c	g	c	a	c	g	c	g	c	c	
Border length $b[j]$:	-1	0	0	1	0	1	2	3	2	3	1

$b[6]=2$ (The p-segment “cgcacg” has a widest border of length 2)

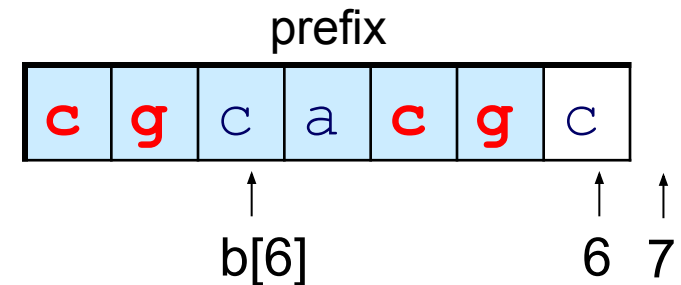
Computation of $b[7]$ from $b[6]$:

$$j = 6$$

Is $P[j] == P[b[j]]$?

Yes: the border can be extended to 'cgc'

Therefore, $b[7] = b[6] + 1 = 3$.



Example (cont'd)

j :	0	1	2	3	4	5	6	7	8	9	
<i>Pattern</i> :	c	g	c	a	c	g	c	g	c	c	
Border length $b[j]$:	-1	0	0	1	0	1	2	3	2	3	1

Computation of $b[8]$ from $b[7]$:

$j = 7$

Is $P[j] == P[b[j]]$?

No: the border 'cgc' cannot be extended.

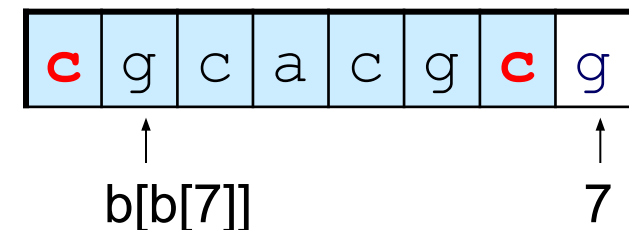
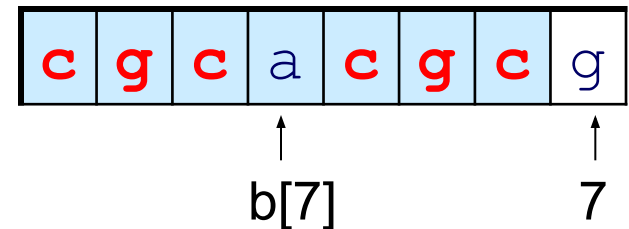
Does the border have a sub-border? If so, can it be extended? i.e.,

Is $P[j] == P[b[b[j]]]$? ($b[b[7]] = b[3] = 1$)

Yes. We extend the sub-border.

Therefore,

$$b[8] = b[b[7]] + 1 = 2.$$

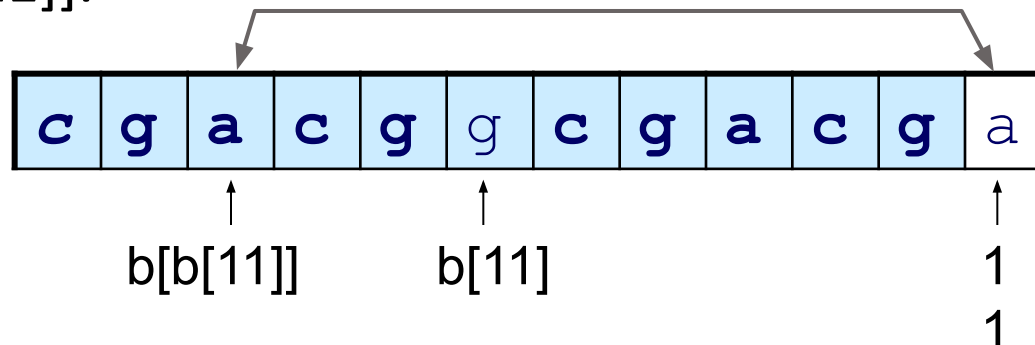


Preprocessing Patterns

Another Example:

	0	1	2	3	4	5	6	7	8	9	10	11	12
	c	g	a	c	g	g	c	g	a	c	g	a	
$b[j]$	-1	0	0	0	1	2	0	1	2	3	4	5	?

To compute $b[12]$, we first check if the previous border can be extended: Is $P[11] == P[b[11]]$? If not we compare the character at position $b[b[11]]$.



They match. So $b[12] = b[b[11]] + 1 = 3$

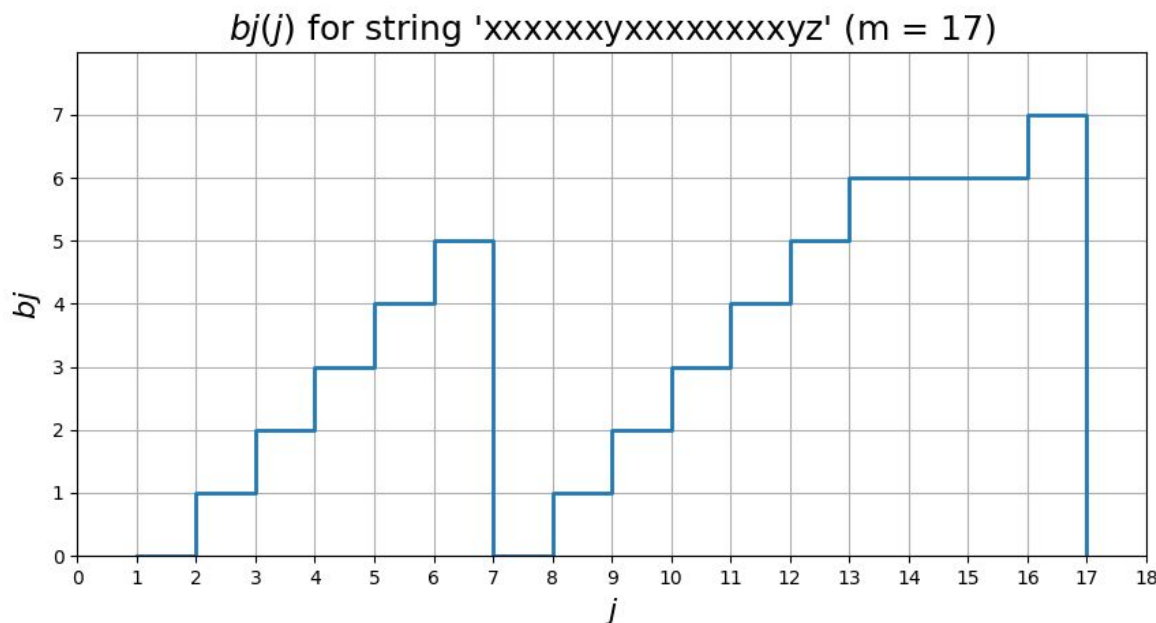
Efficient Computation of Prefix Borders

Although the explanation and proof is complicated, the code is quite simple:

```
def border_lengths(p):  
    """ Computes lengths of p-segment borders """  
    b = [0]*(len(p) + 1)  
    b[0:2] = [-1, 0]  
    j, bj = 1, 0  
    while j < len(p):  
        while bj >= 0 and (p[j] != p[bj]):  
            bj = b[bj]  
        j, bj = j+1, bj+1  
        b[j] = bj  
    return b
```

Preprocessing - Complexity

- The outer while-loop executes exactly $m-1$ times.
- The inner loop can decrease the value of b_j at most as often as it has been increased previously by the outer loop.
- The overall number of executions of the nested loops follow a staircase pattern with an overall complexity of $O(m)$.

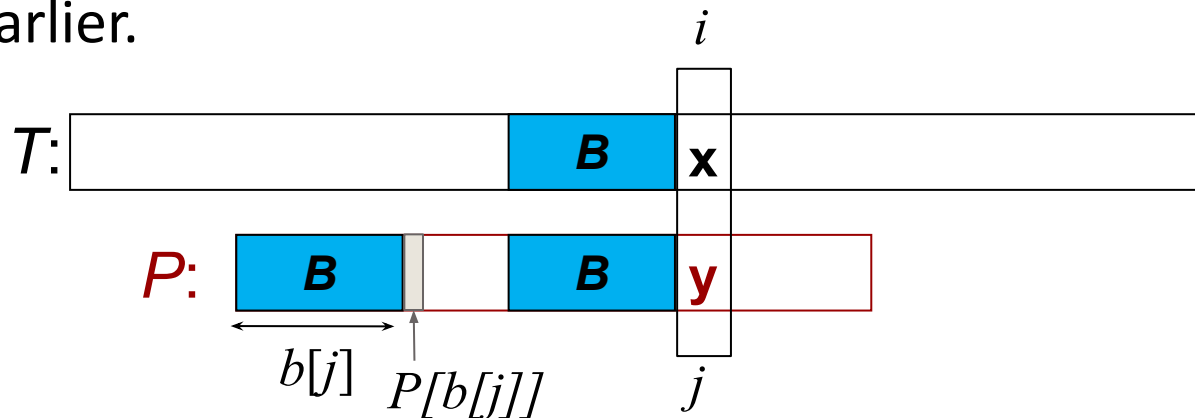


"Amortised
complexity"
(See COSC261)

Shifting Pattern Using P-segment Borders

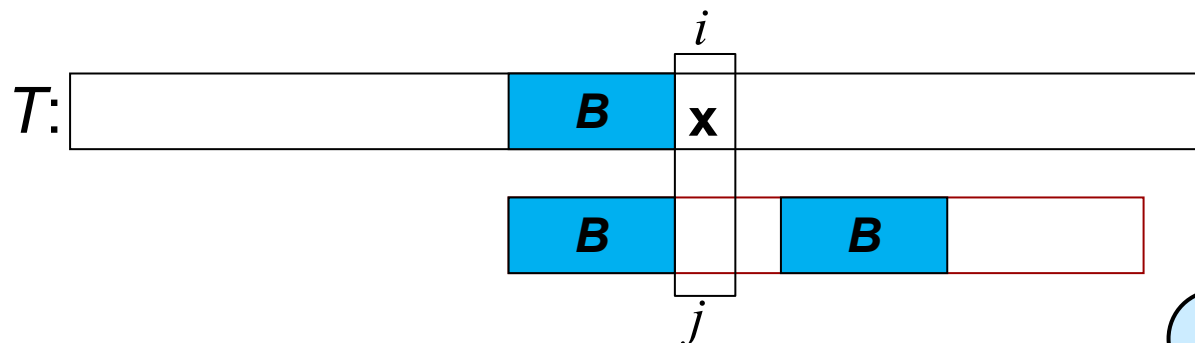
- We now consider the procedure for shifting the pattern as previously shown earlier.

- $T[i] \neq P[j]$



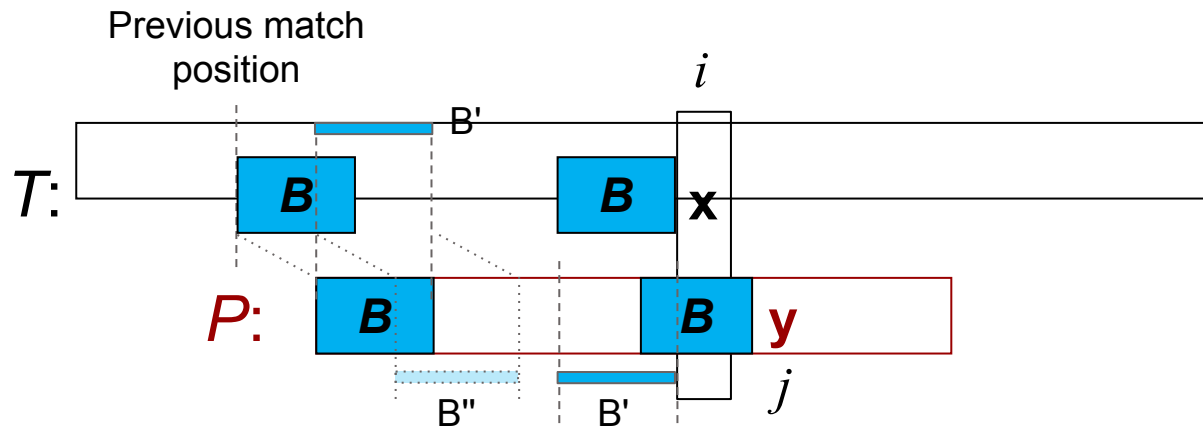
- We change the value of j to $b[j]$ thereby **effectively** shifting the pattern as shown below. **i does not change.**

- $T[i]$ and $P[j]$ are now compared.



Proof that this is the minimum shift

- Suppose we try some smaller shift in P and get a match



- Sections B' in T and P must match B as well.
- Hence the B/B' union is a border of the p-segment $P[j]$
- But it's wider than just B
- So B *wasn't* the widest border of the p-segment $P[j]$
- CONTRADICTION \Rightarrow aligning borders *is* the minimum shift

KMP Algorithm (Code)

```
b = border_lengths(pattern)

i, j = 0, 0
while i < len(text):
    while j >= 0 and text[i] != pattern[j]:
        j = b[j]
    i, j = i+1, j+1
    if j == len(pattern):
        # Match found. Print position.
        print (i-j)
        j = b[j]
```

KMP Example with Visualiser

<https://lobb.nz/stringmatchvisualiser/stringmatching.html>

Use the string match visualiser on the following:

Text: cgacggcgacggcgggcgaccgacggcgacgac

Pattern: cgacggcgacga

Border table (slide 33):

j	0	1	2	3	4	5	6	7	8	9	10	11	12
pattern	c	g	a	c	g	g	c	g	a	c	g	a	
$b[j]$	-1	0	0	0	1	2	0	1	2	3	4	5	3

KMP Algorithm visualised

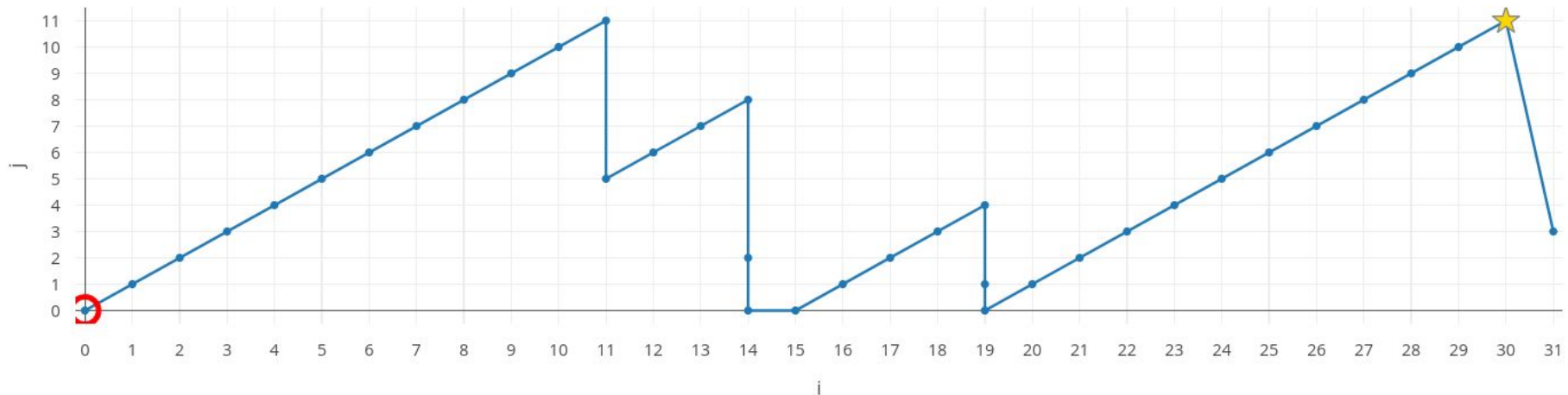
i:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Text:	c	g	a	c	g	g	c	g	a	c	g	g	c	g	g	c	g	a	c	c	g	a	c	g	g	c	g	a	c	g	a	c
Pattern:	c	g	a	c	g	g	c	g	a	c	g	a																				

Previous

Next


[Show state table](#)

Comparison Trajectory

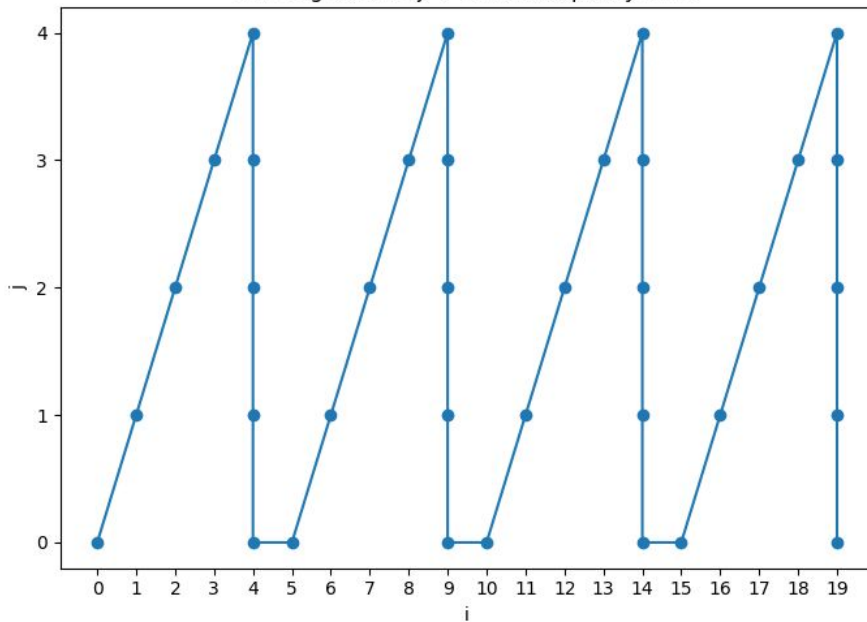


KMP: Another example (worst case)

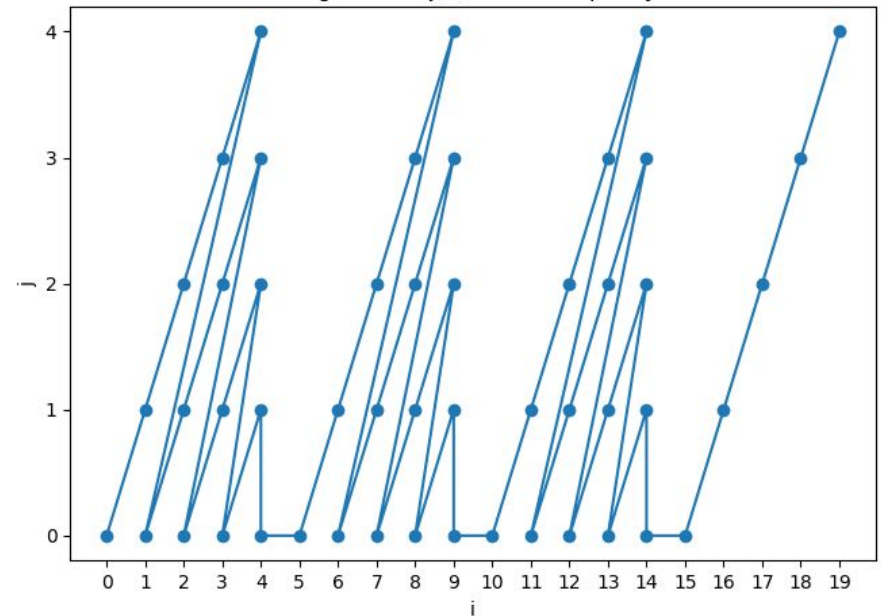
XXXXXOXXXXXOXXXXXOXXXXXO

XXXXXX

KMP algorithm: (j, i) for char equality tests



Naive algorithm: (j, i) for char equality tests



KMP Algorithm Complexity

- The previous examples show that comparisons form a “sawtooth” pattern. There are two cases:
 - $T[i] = P[j]$. Here i and j are both incremented in the outer while loop.
 - $T[i] \neq P[j]$. Here the value of j is repeatedly reduced to $b[j]$ in the inner loop, effectively moving the pattern towards the right. i does not change (until $j = -1$ when the inner loop terminates).
- Each "tooth" has at most as many points in the vertical bit as in the sloped bit, so at most $2n$ comparisons in total.
- Preprocessing requires $O(m)$ comparisons.
- The overall complexity of KMP algorithm is $O(m+n)$

Boyer-Moore Algorithm

- A very efficient string matching algorithm
- Often implemented in search and replace functions in text editors.
- Preprocesses the search string to derive information about its structure
- Compares the search pattern from **right to left** of the search string
- Uses two types of heuristics:
 - “Bad character” heuristics
 - “Good suffix” heuristics

Bad Character Heuristics

If a mismatch occurs for the **rightmost character** $P[m-1]$ of the search pattern, and if the corresponding character in the text does not appear in the pattern at all, then the whole pattern can be shifted by m positions.

agtacg**t**gctcaaccgagtatagccgagatacg

gccgaga



gccgaga



Shift

Boyer-Moore Complexity (Best Case)

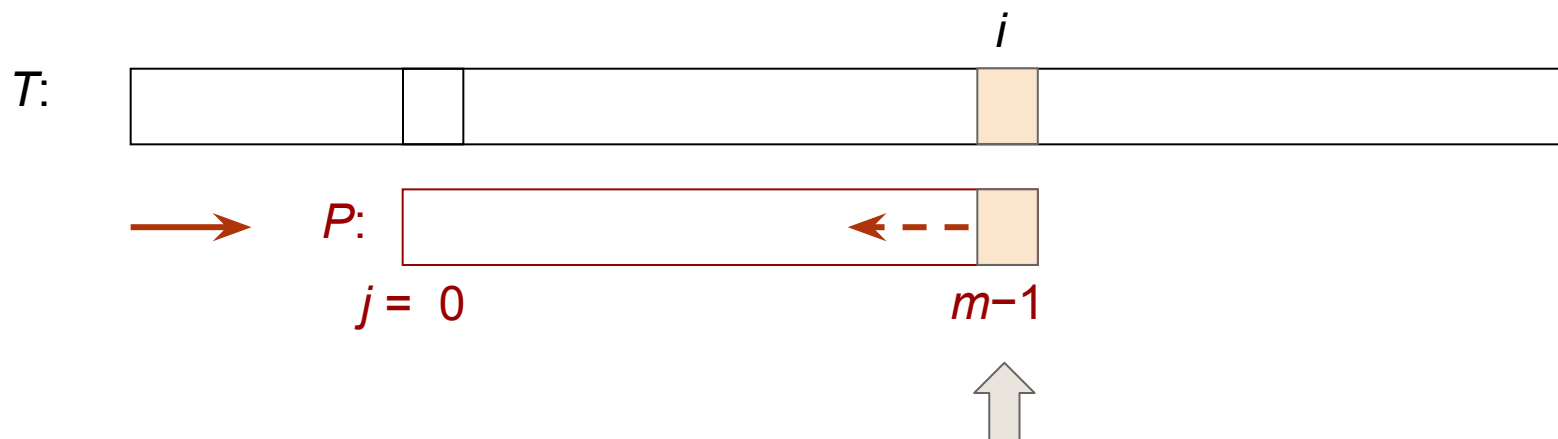
If the first compared text symbol does not occur in the pattern for each attempt, the number of comparisons would be of the order of $O(n/m)$ (Sub-linear complexity!)

Example:

aaaaaa**b**aaaaaa**b**aaaaaa**b**aaaaaaabbbbbbbbbbbb
aaaaaa▲
 aaaaaa▲
 aaaaaa▲

Comparison of characters

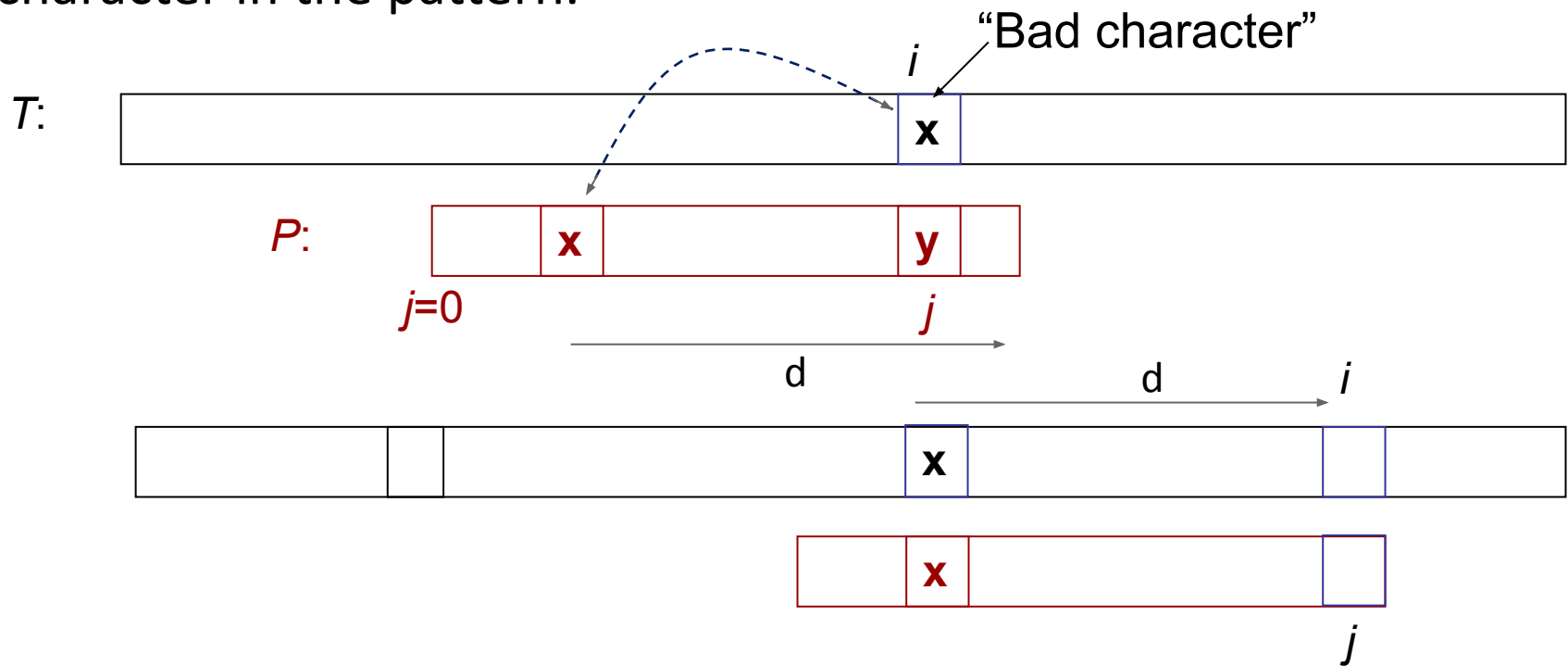
- In Boyer-Moore algorithm, we always start comparisons from the rightmost end of the pattern and work backwards.
- The pattern is always moved from left to right.



Start comparing here and move left

Bad Character Heuristics

If a text symbol that causes a mismatch at $P[j]$ is found somewhere in the pattern, then the pattern can be shifted so that the text symbol aligns with the rightmost occurrence of that character in the pattern.

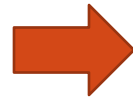


Restart comparisons at $i = i + d, j = m - 1$ where d is the distance of 'x' from the end of the pattern.

Shift array *delta1* ('d' in previous slide)

We create an array “delta1[]” that contains the distance **from the end of the pattern** of the last (rightmost) occurrence of each character (m , if the character does not appear in the pattern).

Example: Pattern: gccgaga
 $m = 7$



Symbol s	'a'	'c'	'g'	't'
delta1[s]	0	4	1	7

agtacgtgctcaac**ccga**gtatagccgagatacg

gccgaga

gccgaga

gccgaga

gccgaga

Shift distance = 4

Shift distance = 1

Shift distance = 7

Computation of “delta1” array

Code for computing the values of “delta1” array:

pat = “agccgcaga”



Symbol s	'a'	'c'	'g'	't'
delta1[s]	0	3	1	9

```

LEN_ALPHABET = 256 # Assume ASCII
def bad_char_shifts(pattern):
    m = len(pattern)
    delta1 = LEN_ALPHABET * [m]
    for i, c in enumerate(pattern):
        delta1[ord(c)] = m - i - 1
    return delta1

```

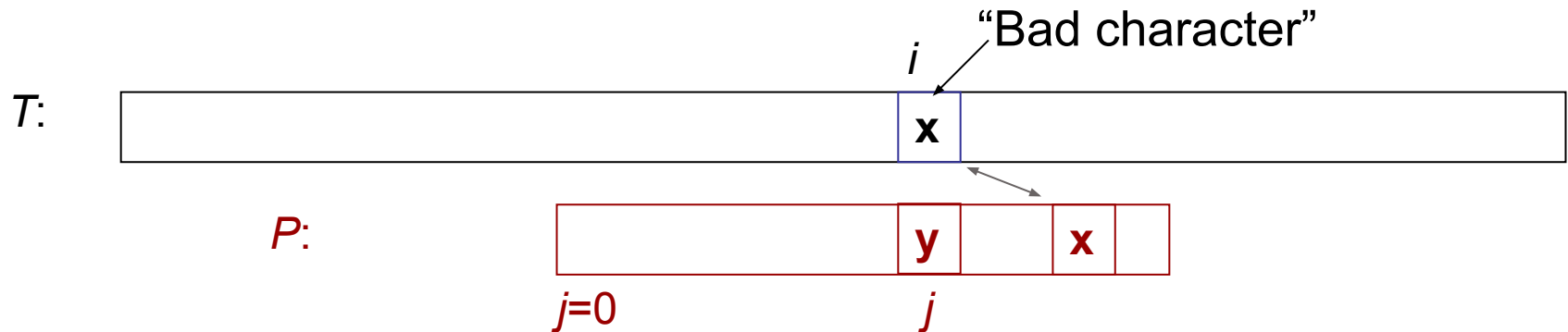
Complexity = $O(m + |\Sigma|)$



Size of alphabet (4 for a DNA sequence if encode *a*, *c*, *g*, *t* as 0, 1, 2, 3)

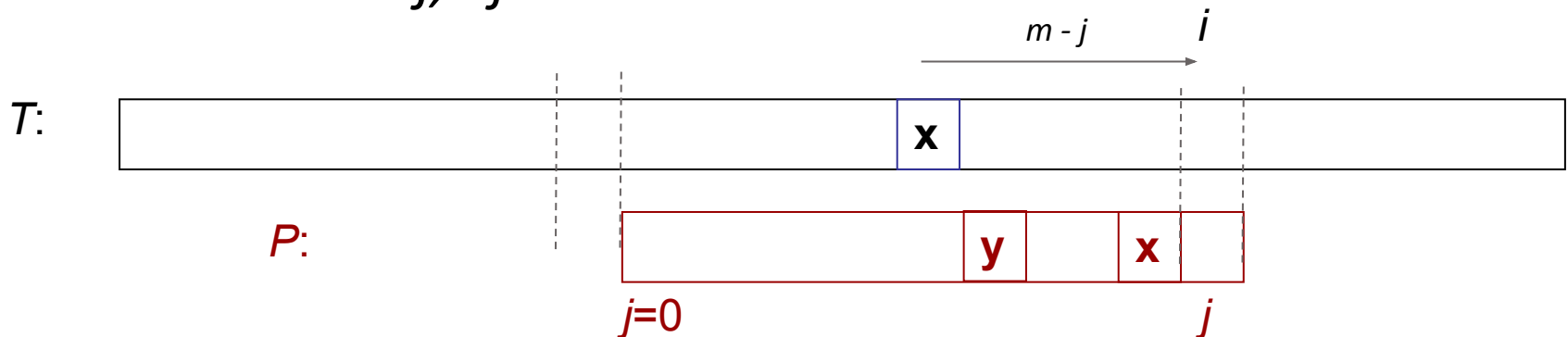
A complication

What if the character we want to match appears *after* $P[j]$?



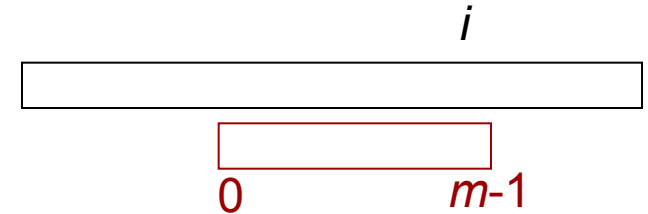
Mustn't slide pattern *back*. Instead just advance it by 1.

Restart at $i += m - j$, $j = m - 1$



Simplified Boyer-Moore (Boyer-Moore-Horspool)

```
def main():
    delta1 = bad_char_shifts(pattern)
    m = len(pattern)
    i = m - 1
    while i < len(text):
        j = m - 1
        while j >= 0 and pattern[j] == text[i]:
            i -= 1
            j -= 1
        if j < 0:
            print(i)
            i += m + 1
        else:
            i += Math.max(m - j, delta1[ord(text[i])])
```



Deals with complication in last slide

delta1 is the increment in *i* **not** the shift

Simplified Boyer-Moore (Boyer-Moore-Horspool)

Watch it in action:

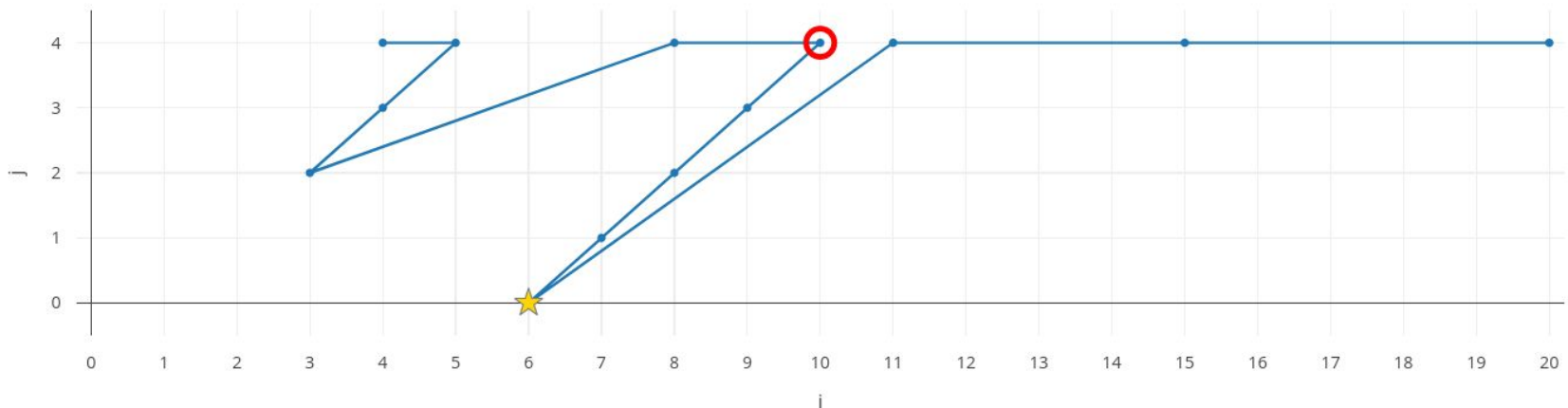
<https://lobb.nz/stringmatchvisualiser/stringmatching.html>

i: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
 Text: t h e m a t h e m a t i c a l t h e o r y
 Pattern: t h e m a

Previous Next

Show state table

Comparison Trajectory



Simplified Boyer-Moore

- For most practical applications, the simplified version of the Boyer-Moore algorithm gives very good results.
- The worst-case running time of the simplified Boyer-Moore algorithm is still $O(nm)$

Example:

aaa

baaaaaa

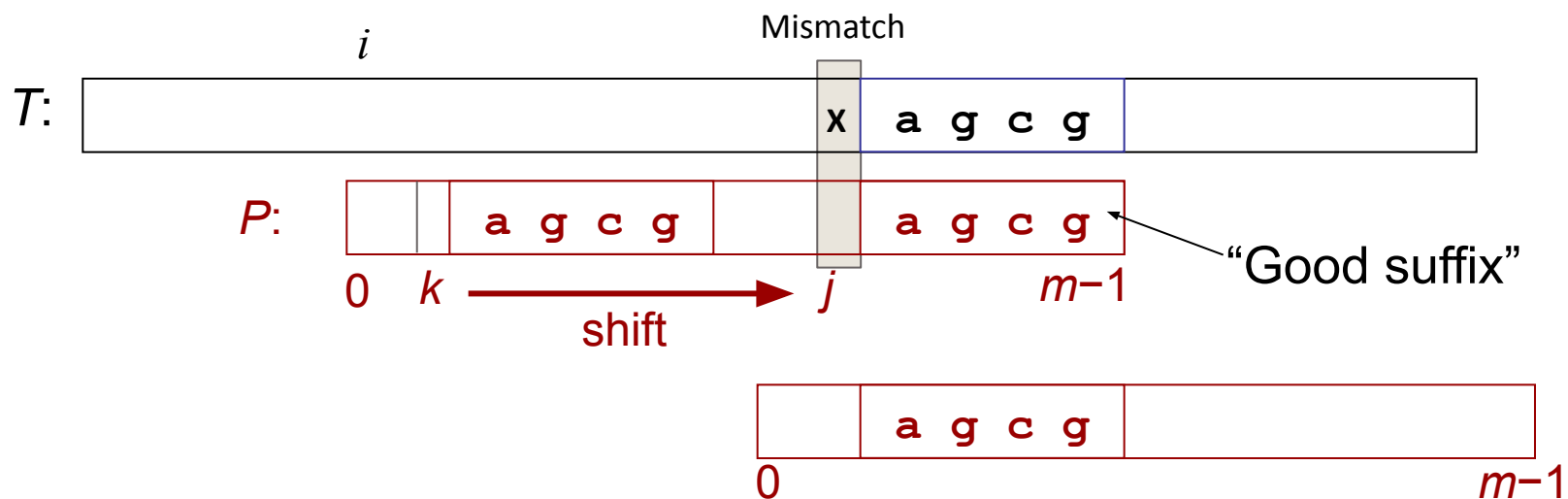
baaaaaa

baaaaaa

Good Suffix Heuristic

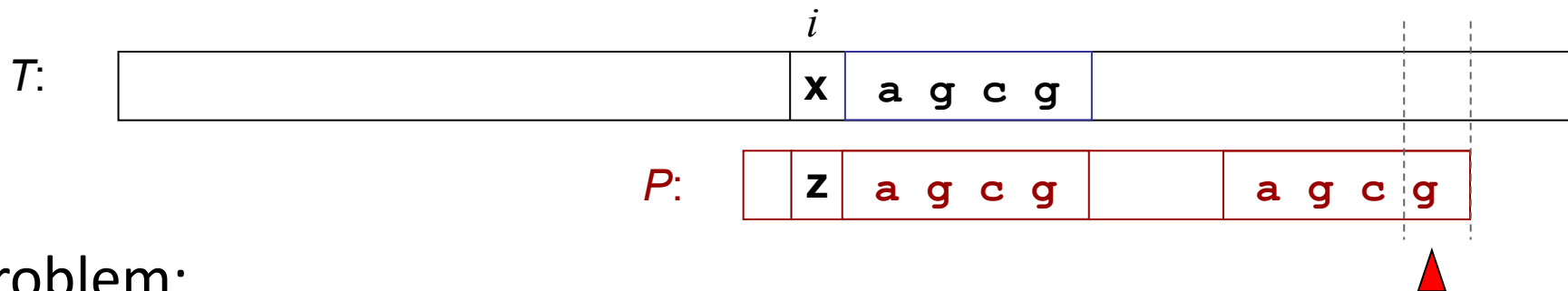
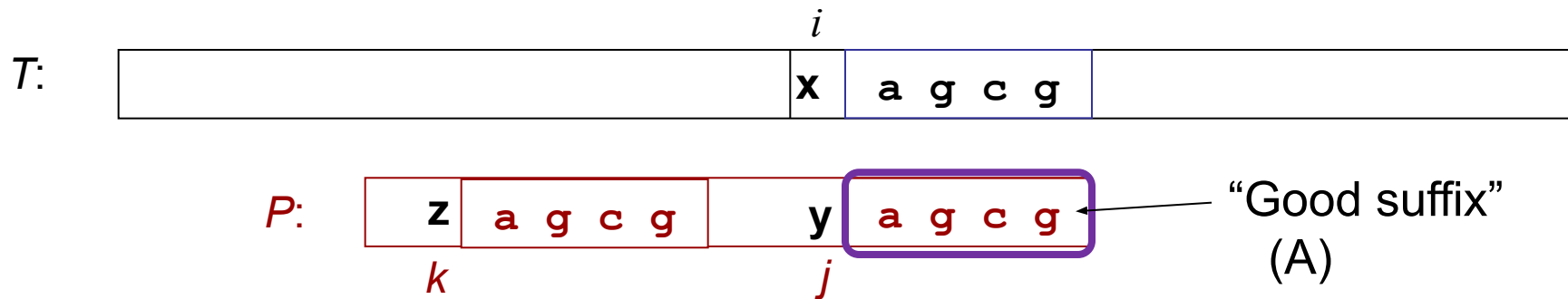
Assume that at the current position, a suffix of P (“good suffix”) matches with the corresponding characters in T .

- If x is in this suffix (e.g. $x == 'c'$), we cannot use $\text{delta1}[x]$ to shift the pattern. Simplified Boyer-Moore uses a shift distance = 1.
- BUT if the matching suffix appears elsewhere within the pattern, then we may be able to shift the pattern by a distance > 1 .



Note: $P[k]$ must be different from $P[j]$, or shifting the pattern will cause another mismatch.

Good Suffix Heuristic

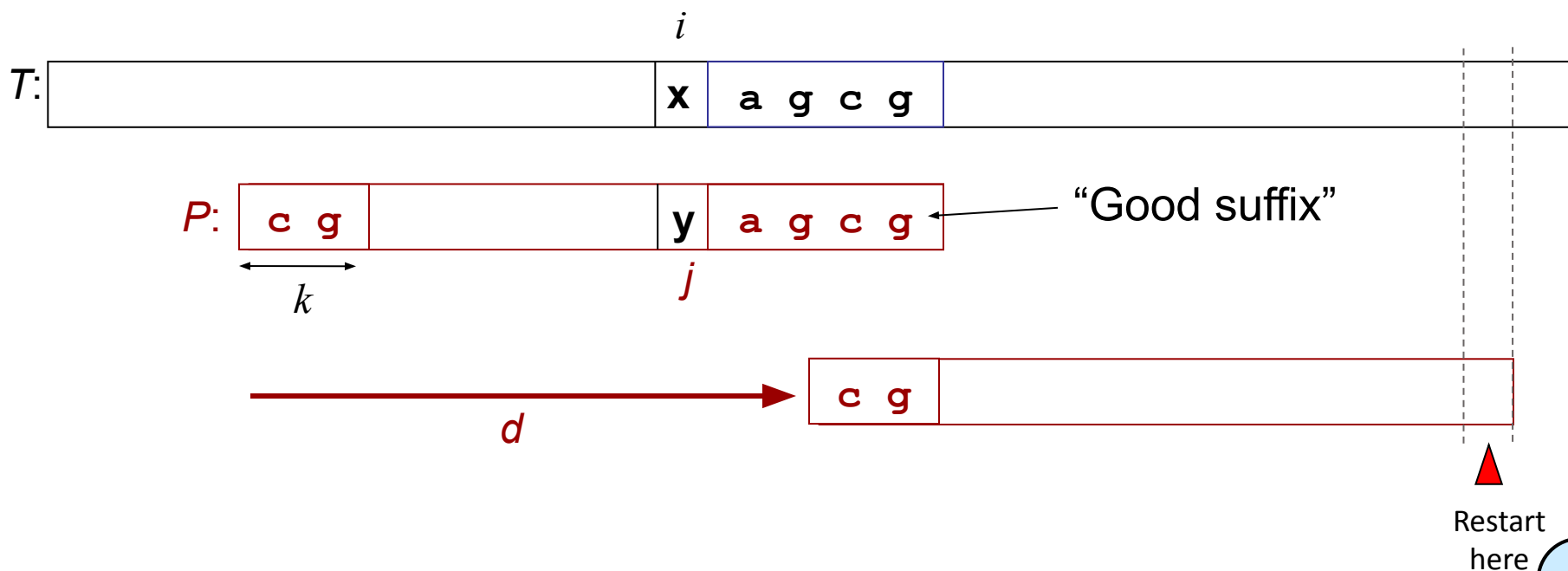


Problem:

- Does the suffix A exist earlier in the pattern (before j)?
- If so, and if $P[k] \neq P[j]$, then shift distance = $j - k$
 - If there are multiple such occurrences, choose the last one

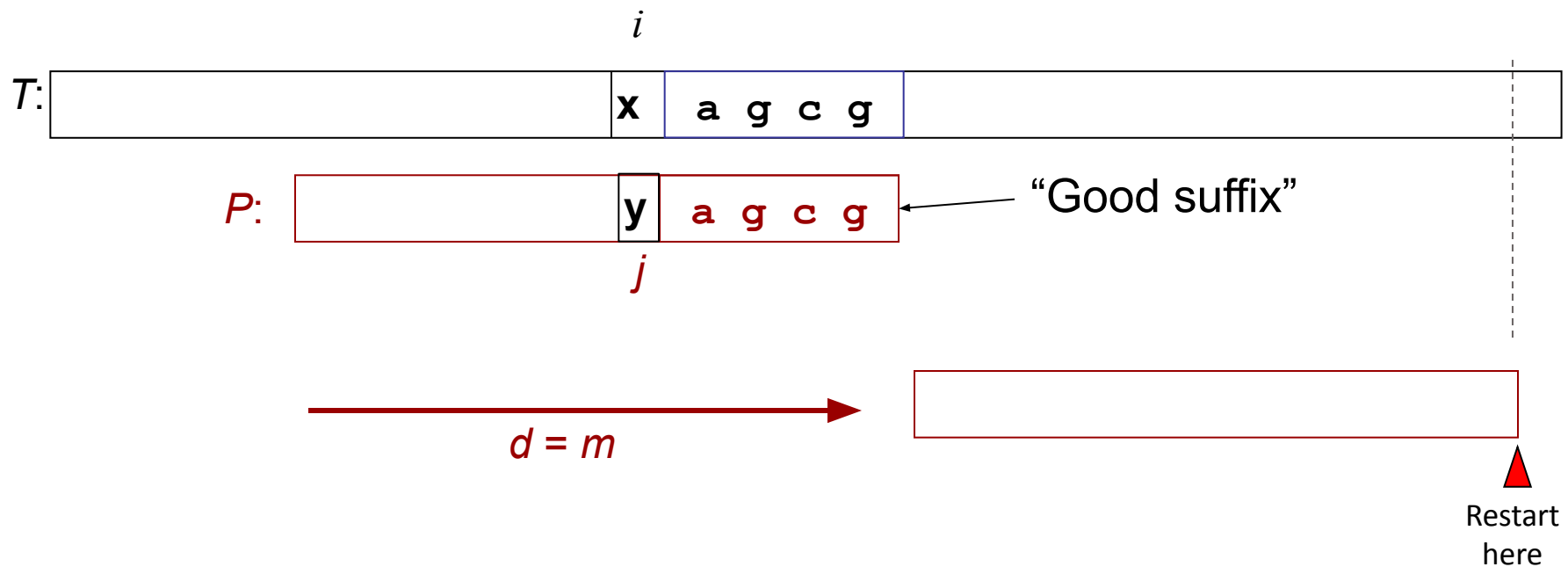
Good Suffix Heuristic

If only a part of the good suffix appears at the beginning of the pattern (the pattern has a border of length k), we can shift the pattern by a distance $d = m - k$

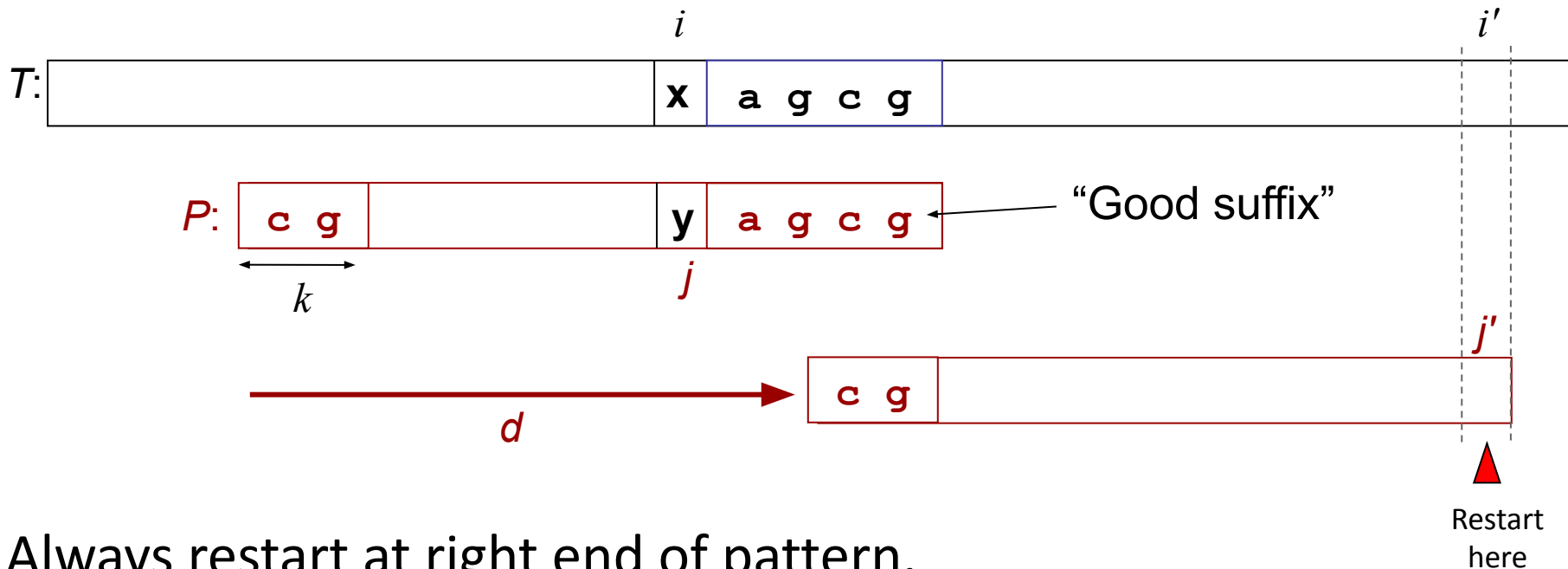


Good Suffix Heuristic

If the good suffix does not appear anywhere else in the pattern and if the pattern does not have a border, then the pattern can be shifted through its entire length ($d = m$)



Updating i and j on a pattern shift



Always restart at right end of pattern.

$$j' = m - 1$$

$$\text{Old pattern shift} = i - j$$

$$\text{New pattern shift} = i' - j' = \text{old pattern shift} + d = i - j + d$$

$$\text{Hence } \Delta i = i' - i = m - 1 - j + d \quad \leftarrow \text{Call this } \textit{delta2}[j]$$

Good Suffix Heuristic: Example

$\text{delta2}[j]$ gives the update to i when a bad character is found at position j : $\Delta i = i' - i = m - 1 - j + d$, where d is the pattern shift.

Good suffix = $P[j+1:]$

Example:

$m = 12$

j	0	1	2	3	4	5	6	7	8	9	10	11
$\text{pat}[j]$	g	a	t	c	a	c	a	c	a	t	c	a
d	12	12	12	12	12	12	12	7	12	3	10	1
$\text{delta2}[j]$	23	22	21	20	19	18	17	11	15	5	11	1

Good suffix: 'ca', $d = 3$

We won't bother with the code to compute these shifts.

Earlier occurrence of tca is preceded by 'a' so not useful. See note slide 54.

Full Boyer-Moore Algorithm

```
delta1 = bad_char_shifts(pattern)
delta2 = good_suffix_shifts(pattern) # Not in course

m = len(pattern)
i = m - 1
while i < len(text):
    j = m - 1
    while j >= 0 and pattern[j] == text[i]:
        i -= 1
        j -= 1
    if j < 0:
        print(i)
        i += m + 1
    else:
        i += max(delta2[j], delta1[ord(text[i])])
```

Compare with
simplified B-M code

Full Boyer-Moore

Watch it in action:

<https://lobb.nz/stringmatchvisualiser/stringmatching.html>

i: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22

Text: t h e m a t h e m a t i c a l t h e o r y

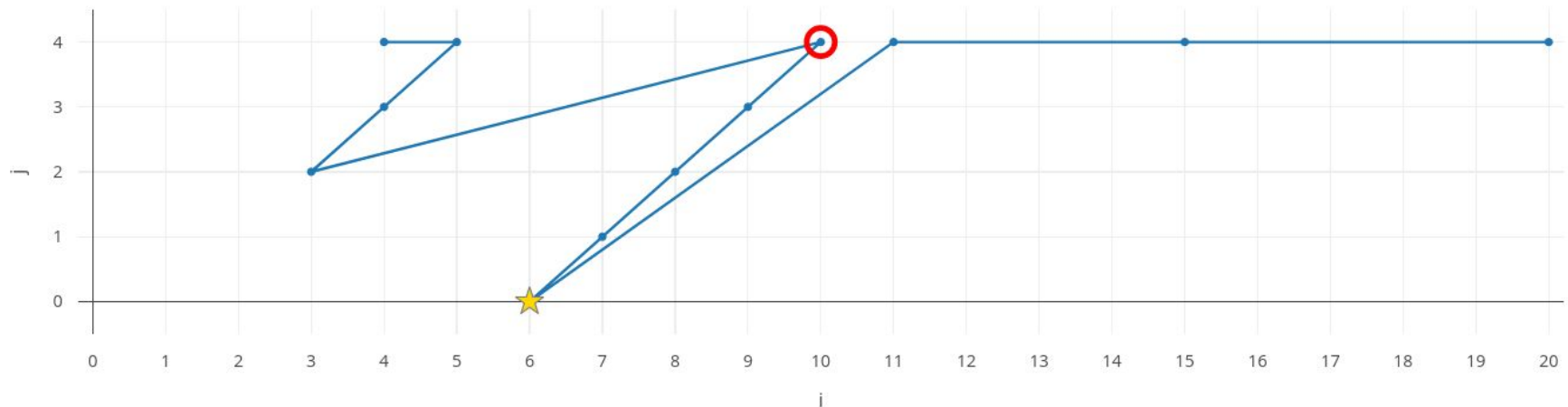
Pattern: t h e m a

Previous

Next

Show state table

Comparison Trajectory



Compare with Simplified Boyer-Moore trajectory. Explain the difference!

Full Boyer-Moore Complexity

- Worst-case time complexity of the full Boyer-Moore algorithm is $O(n+m)$ unless there are lots of hits.
 - Can then be $O(nm)$, e.g. find pattern *aaaaaa* in a string of one thousand *a*'s
- Boyer-Moore is an efficient algorithm for searching natural language text, particularly with large patterns.

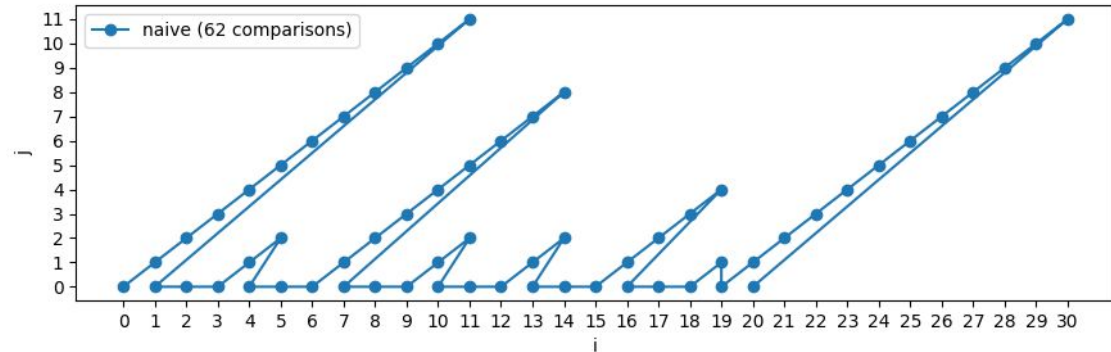
	Best Case	Worst Case
Naïve	$O(n)$	$O(nm)$
KMP	$O(n+m)$	$O(n+m)$
BM	$O(n/m)$	$O(nm)$

Further refinements are possible to B-M to achieve $O(n + m)$ in worst case, namely when there are lots of hits. See Wikipedia (Galil rule).

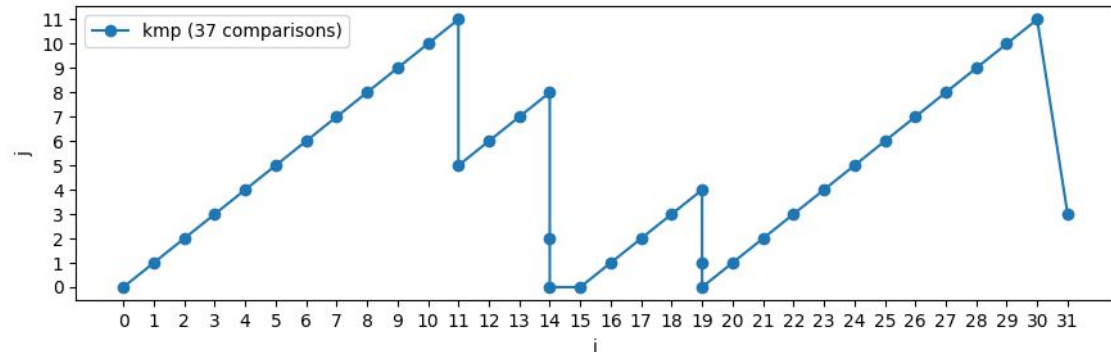
Comparison on DNA example

search("cgacggcgacga", "cgacggcgacggcgggcgaccgacggcgacgac")

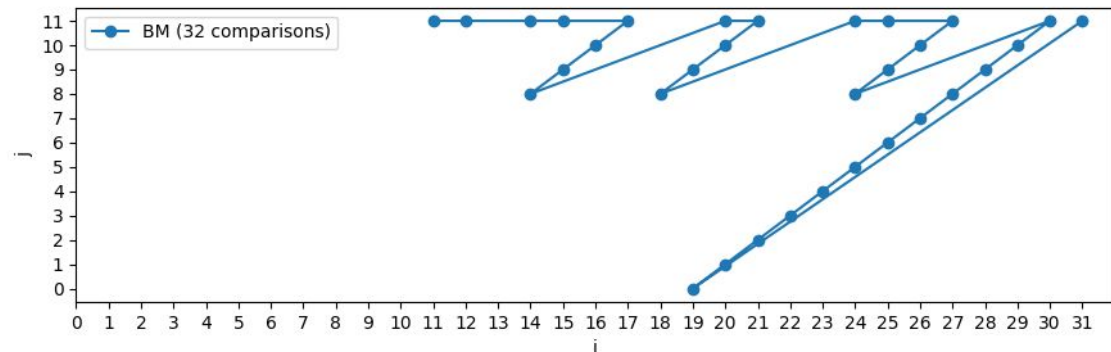
Naive: 62 comparisons



KMP: 37 comparisons



Boyer-Moore: 32 comparisons



Summary

- Naive algorithm is fine for short patterns
- Rabin-Karp is best for searching a large text for a large number of different patterns.
- Boyer-Moore is the best algorithm with large patterns and large texts
 - Boyer-Moore-Horspool is relatively easy to program and gives most of the benefits
 - The full algorithm is difficult and tables are large, especially with a large alphabet
- KMP has best worst case performance; no restriction on alphabet size.
 - And easier to understand and program.