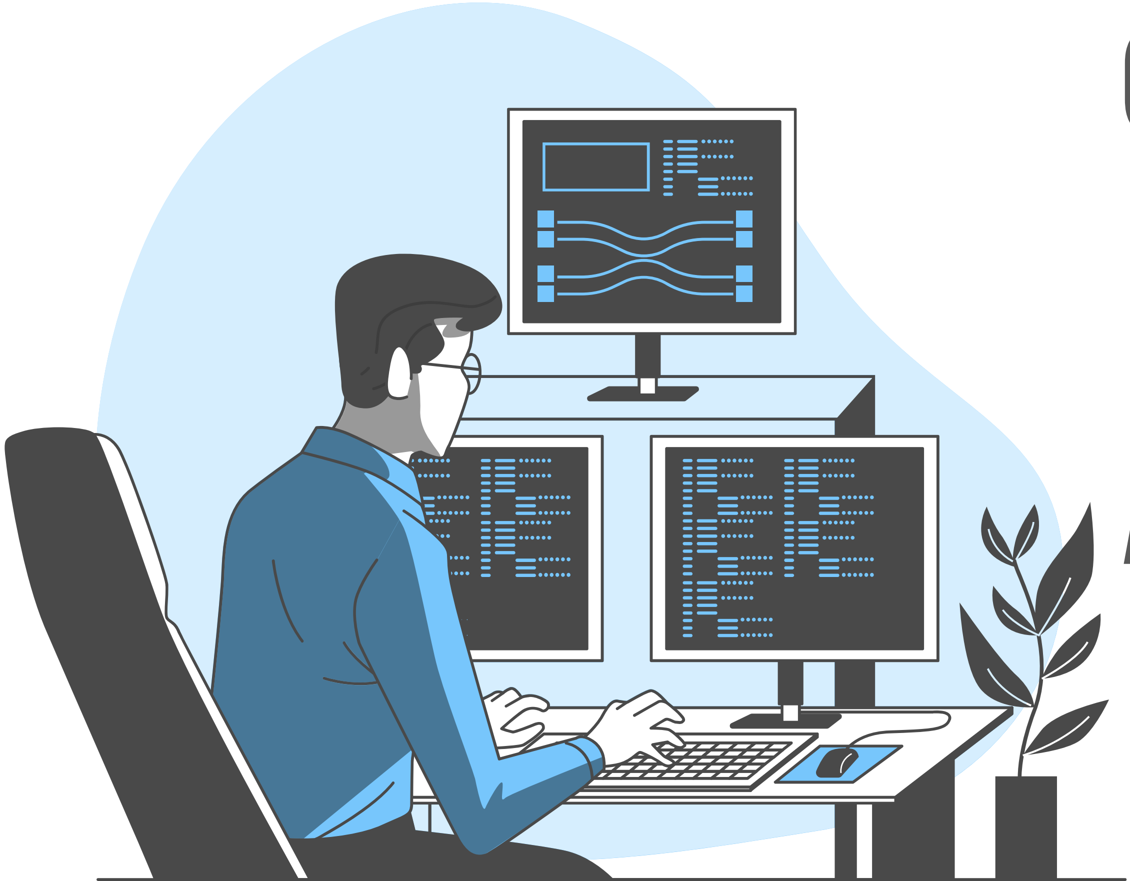


CZ2001 Lab Project 1- Searching Algorithms

Objectives

- To propose algorithms that solve **string searching problems on genome sequences**

- The algorithms should return **positions** of occurrences of a query sequence in the source sequence and the **number of occurrences**

Table of Contents

01

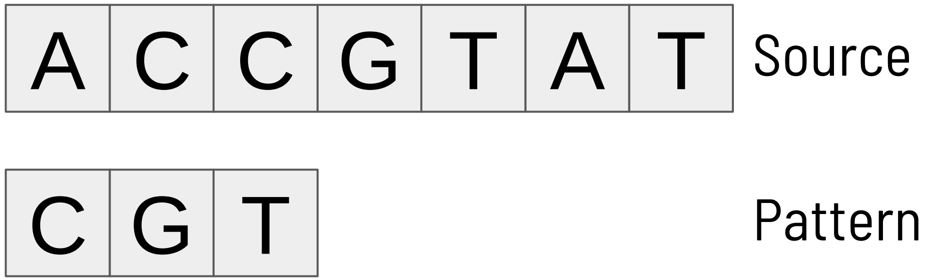
Brute Force

Naive String Searching Algorithm

Concept

- Walk through the source sequence from the beginning till the end

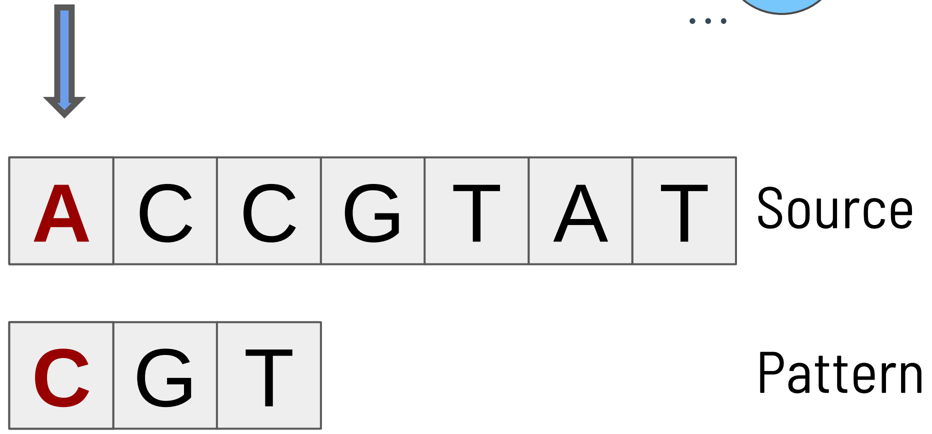
- Check **at each position** if the resulting substring equals the query sequence



Concept

- Walk through the source sequence from the beginning till the end

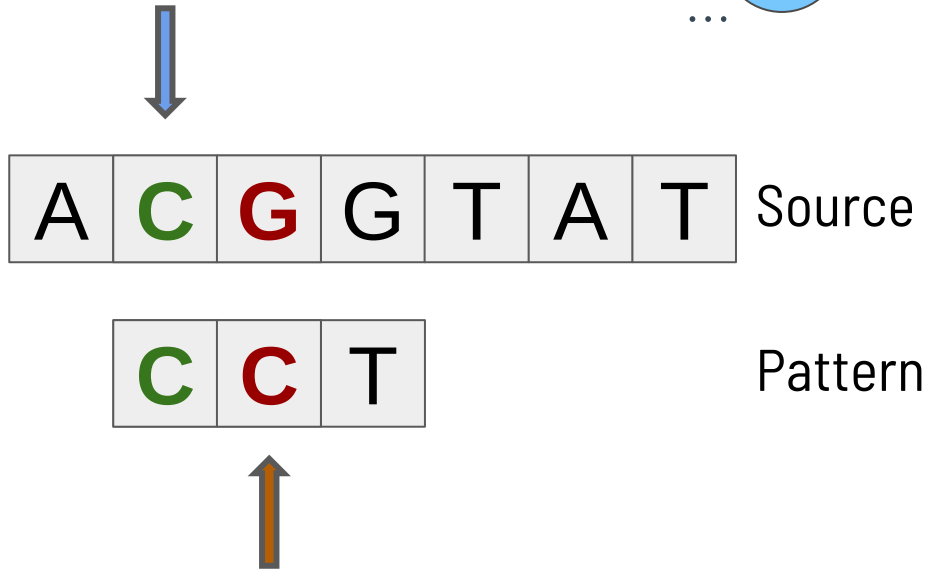
- Check **at each position** if the resulting substring equals the query sequence



Concept

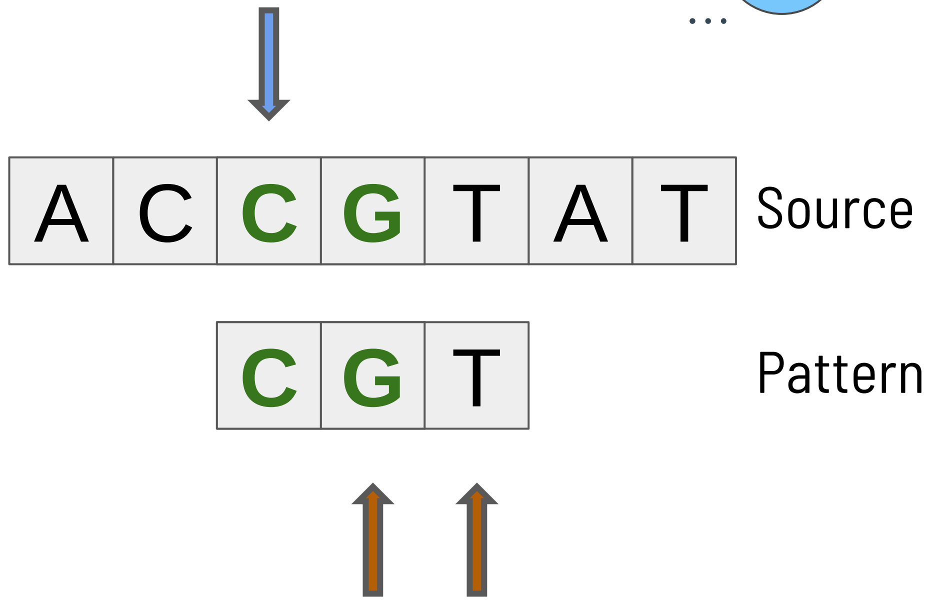
- Walk through the source sequence from the beginning till the end

- Check **at each position** if the resulting substring equals the query sequence



Concept

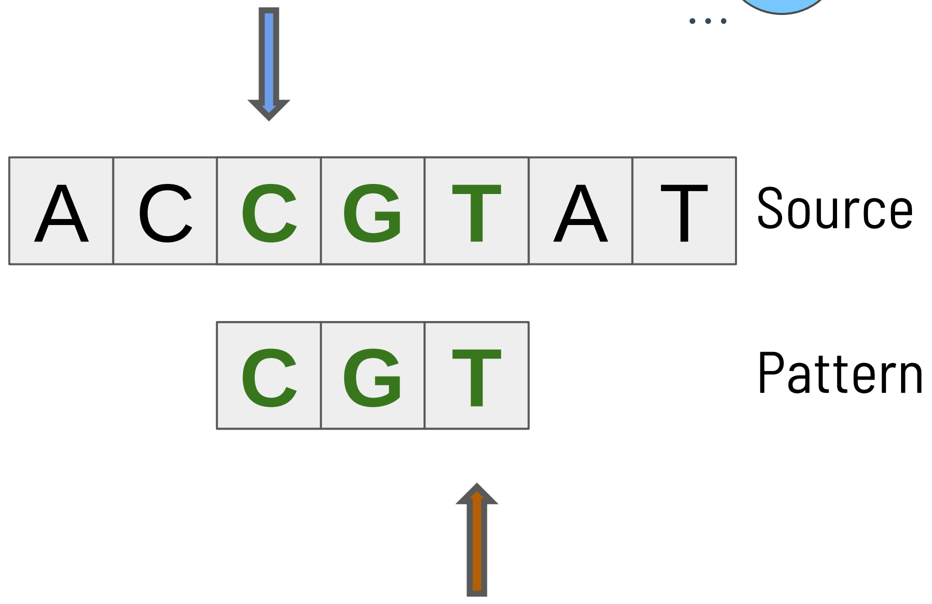
- Walk through the source sequence from the beginning till the end
- Check **at each position** if the resulting substring equals the query sequence



Concept

- Walk through the source sequence from the beginning till the end

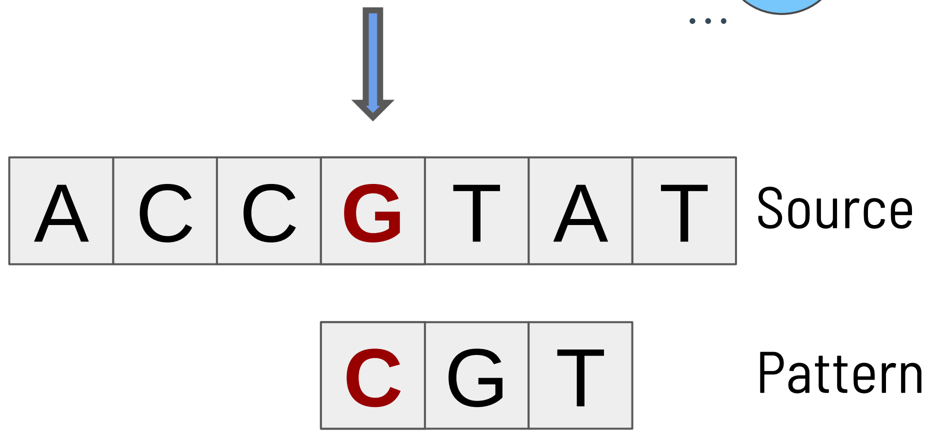
- Check **at each position** if the resulting substring equals the query sequence



Concept

- Walk through the source sequence from the beginning till the end

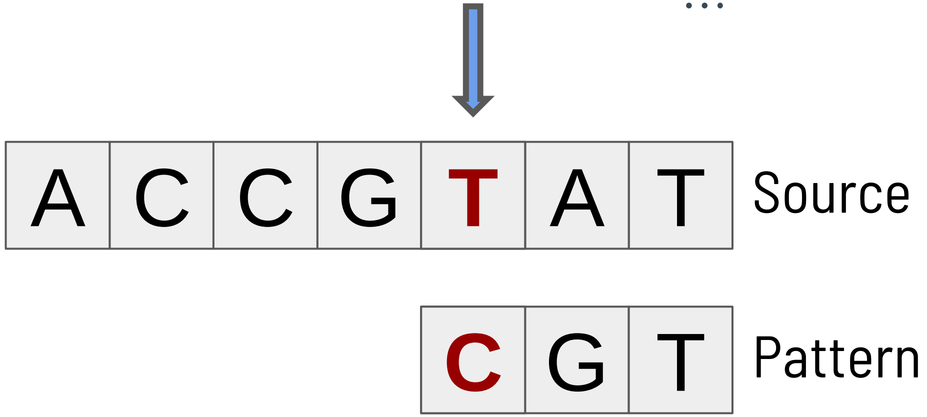
- Check **at each position** if the resulting substring equals the query sequence



Concept

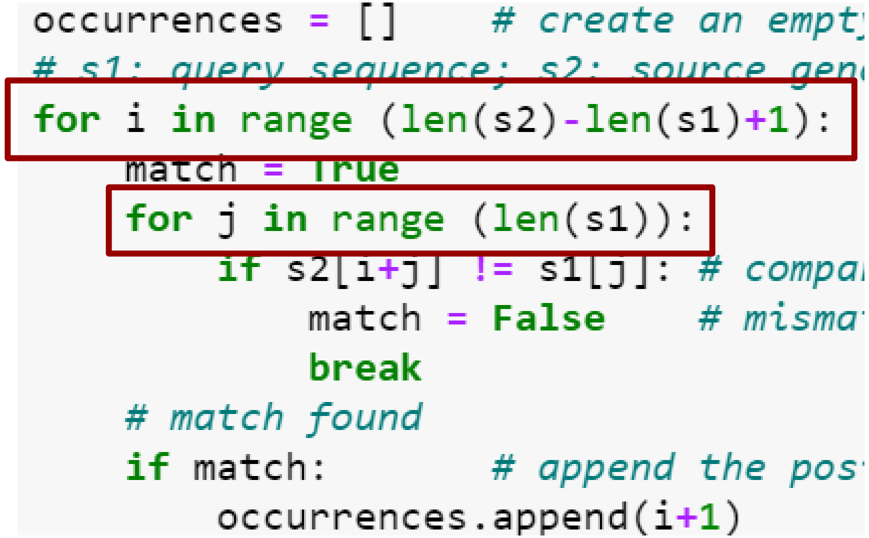
- Walk through the source sequence from the beginning till the end

- Check **at each position** if the resulting substring equals the query sequence



Implementation

- Nested for-loop
  - Outer loop: check along all possible substrings
  - Inner loop: compare characters between the substring and the query sequence



```
occurrences = []    # create an empt
# s1: query sequence; s2: source gen
for i in range (len(s2)-len(s1)+1):
    match = True
    for j in range (len(s1)):
        if s2[i+j] != s1[j]: # compa
            match = False    # misma
            break
    # match found
    if match:           # append the pos
        occurrences.append(i+1)
```

Time Complexity

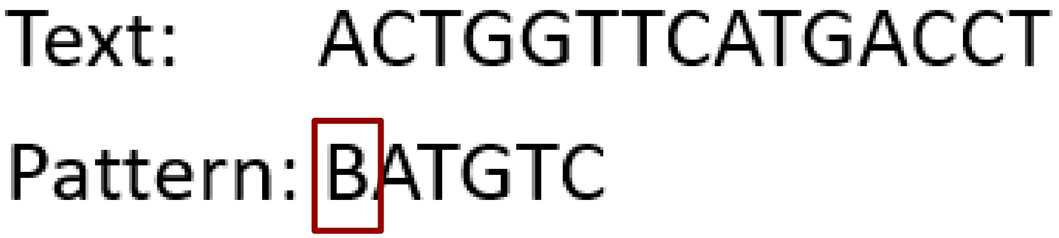
- let **n** denote the length of the source genome sequence, and **m** be the length of the query sequence, where (n>>m)

- To analyze the time complexity of brute force algorithm, we will be looking at the **number of character comparisons**

Time Complexity – Best Case Scenario

- Every first-character comparison between possible substring and query sequence results in a mismatch

- The total number of comparisons is only determined by the number of outer-loop iterations, which is **(n-m+1)**, or **O(n)**



Text: ACTGGTTCATGACCT

Pattern: BATGTC

Time Complexity – Worst Case Scenario

- The total number of comparisons is the **product** of the number of outer-loop comparisons and inner-loop comparisons, which is **m(n-m+1)**, or **O(mn)**

Text: TTTTTTTTTT

Pattern: TTTT

Text: TTTTTTTTTT

Pattern: TTTA

Time Complexity - Average Case Scenario

- Outer loop: it will always be executed (n-m+1) times
- Inner loop: assume each number of comparisons (from 1 to m) is equally likely, expected number of comparisons = (1+m)/2

- The average number of comparisons = (n-m+1)[(1+m)/2] = **O(mn)**

02

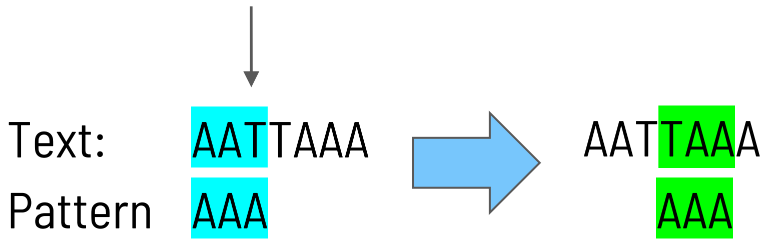
BMH

Boyer Moore Horspool Algorithm

Concept

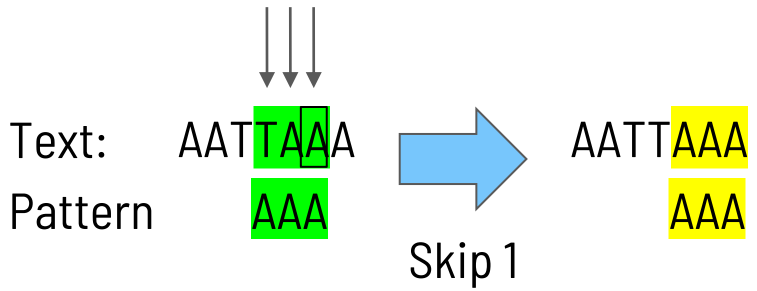
- Reduce searching time by comparing pattern with text sequences from **end to start** of pattern

- **Comparisons between first few characters** can be **skipped** when there is a **mismatch in the last few characters**



Searching in BMH

- **Bad Match Table**: Records number skips to do after mismatch with respect to the **rightmost character compared in the text**



Bad Match Table

A	C	G	T
1	3	3	3

Time Complexity: Preprocessing

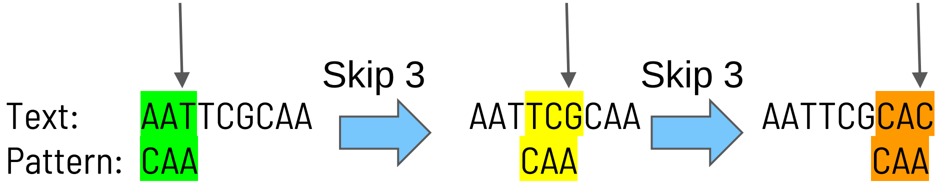
n: Length of text
m: Length of pattern
σ: Number of alphabets

- Counting the number of character comparisons.
- For all cases: $O(m+\sigma)$
    - Assign position to alphabet in Bad Match Table (σ)
    - Calculate number of skips (m)

Time Complexity: Searching

- Best Case Scenario: O(n/m)
  - When the first compared character is always not found in the pattern
  - 1 Comparison and m skips per outer loop

n: Length of text
m: Length of pattern
σ: Number of alphabets
↓: Comparison

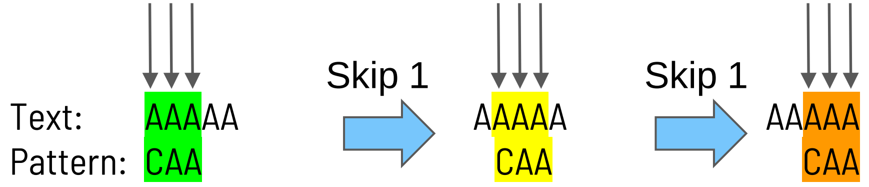


A	C	G	T
1	2	3	3

Time Complexity: Searching

- Worst Case Scenario: O(nm)
  - When all characters in the pattern matches or when all but the last match
  - m comparisons and 1 skips per outer loop

n: Length of text
m: Length of pattern
σ: Number of alphabets
↓ : Comparison



A	C	G	T
1	2	3	3

Time Complexity: Searching

- Average Case Scenario: O(n)

E (Number of comparisons)
= E(Times outer loop executed) * E(Comparisons per outer loop)
= n/E(Skips per outer loop) * E(Comparisons per outer loop)
= n = O(n)

- Assume each skip (from 1 to m) is equally likely
  - E(Skips per outer loop) = (1+m)/2
- Assume each number of comparisons (from 1 to m) is equally likely
  - E(Comparisons per outer loop) = (1+m)/2

n: Length of text
m: Length of pattern
σ: Number of alphabets

03

Modi-KMP

Modified Knuth-Morris-Pratt algorithm

Concept

- Uses two preprocessing techniques to reduce comparisons: **Bad match table(BMT)** & **Longest Prefix-Suffix array(LPS)**

- Implements bad character heuristic within KMP algorithm, which initially use only lps array.

Bad Match Table

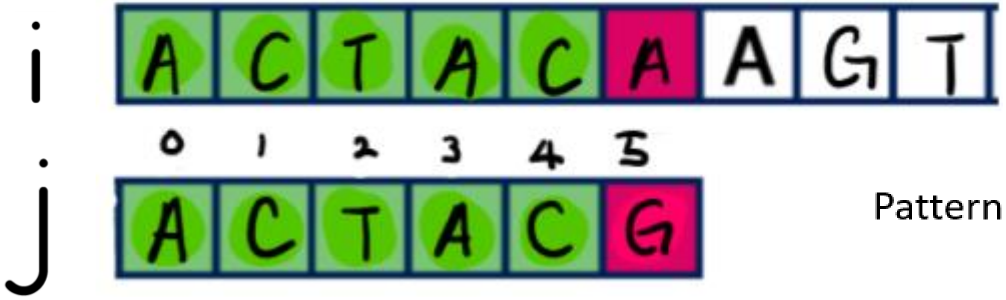
A	C	G	T
1	3	3	3

Pattern: ACTACG

lps arrary

A	C	T	A	C	G
0	0	0	1	2	0

Lps array



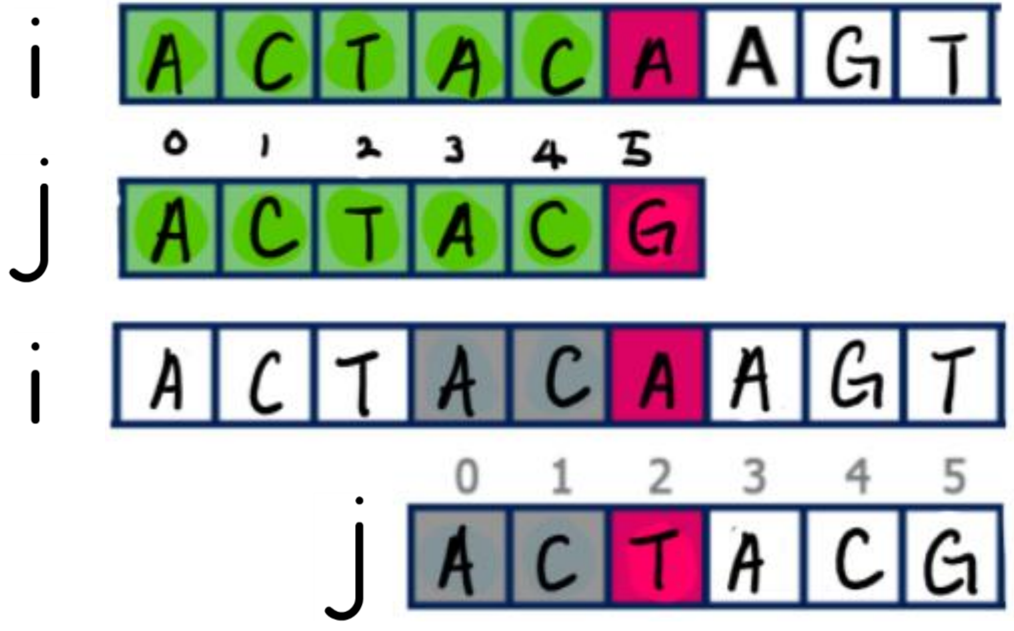
Pattern: ACTACG

lps arrary

A	C	T	A	C	G
0	0	0	1	2	0

i = 5

Lps array



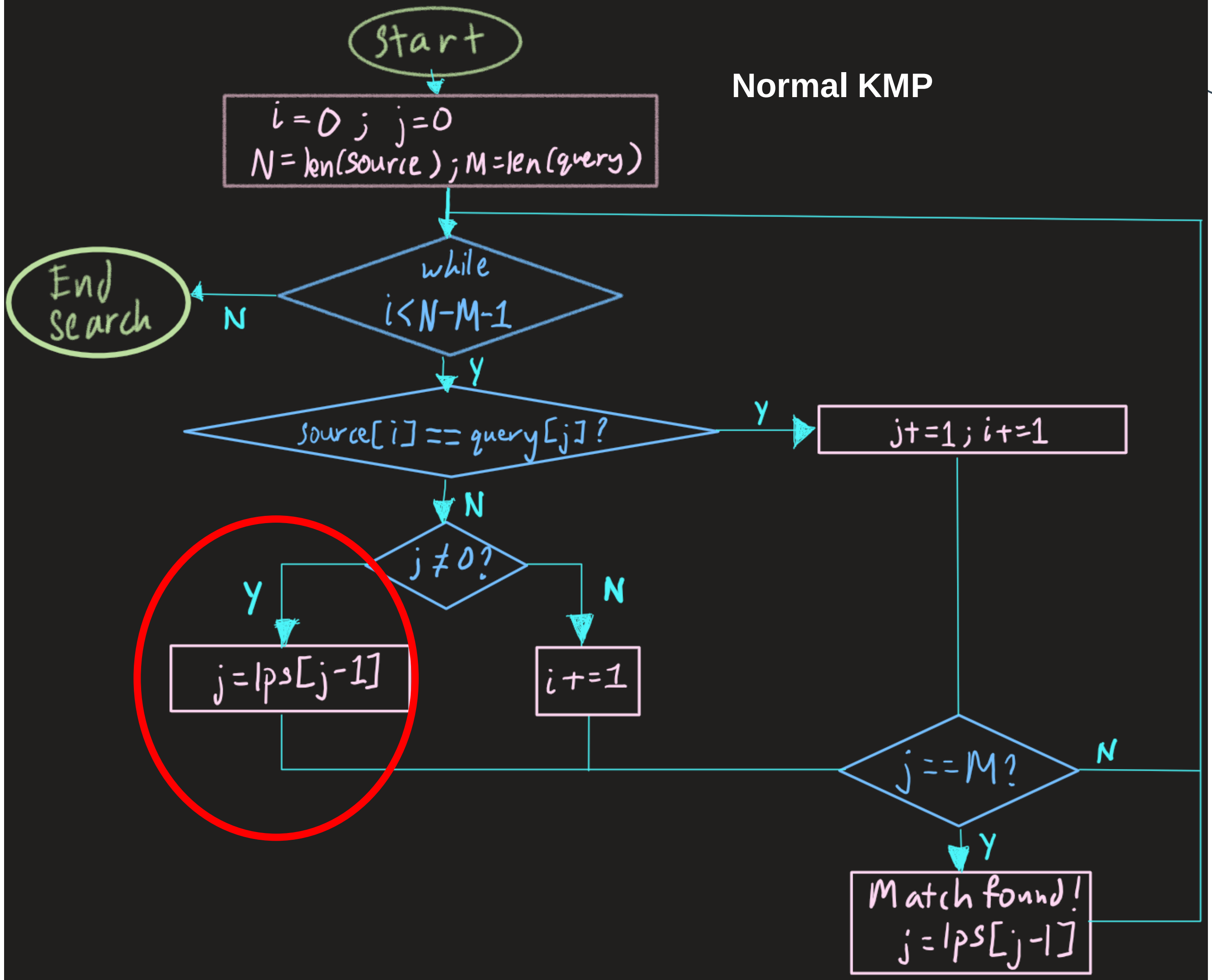
i = 5

Pattern: ACTACG

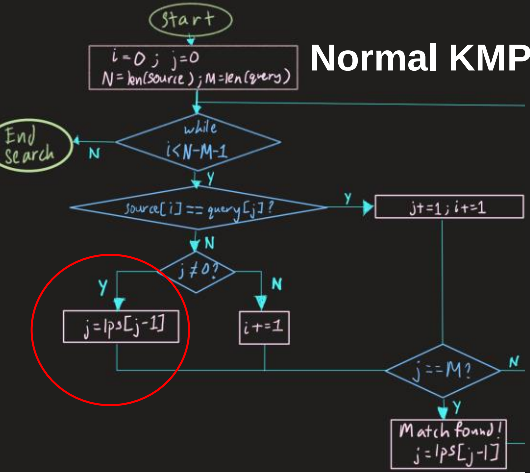
lps array

0	1	2	3	4	5
A	C	T	A	C	G
0	0	0	1	2	0

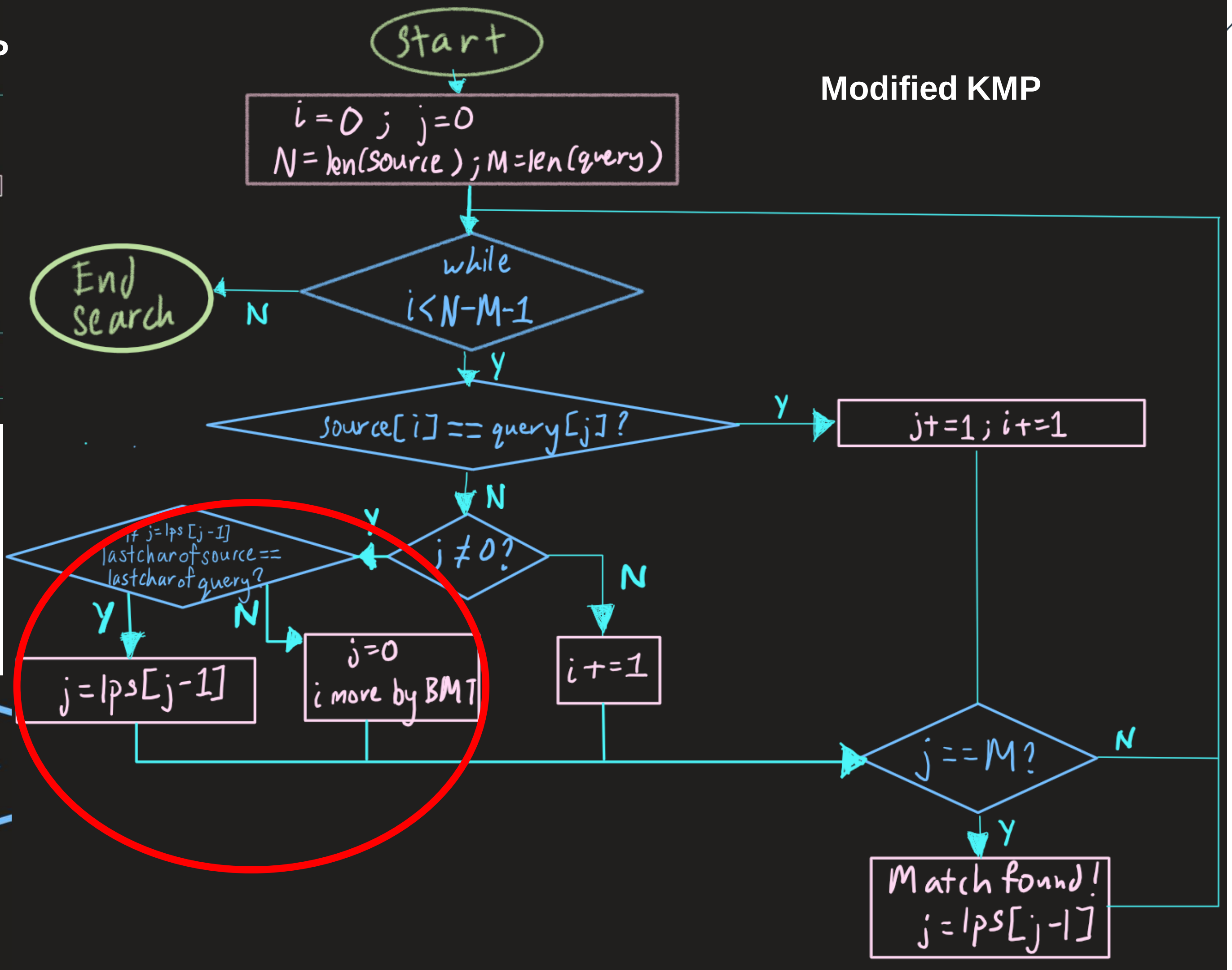
Normal KMP

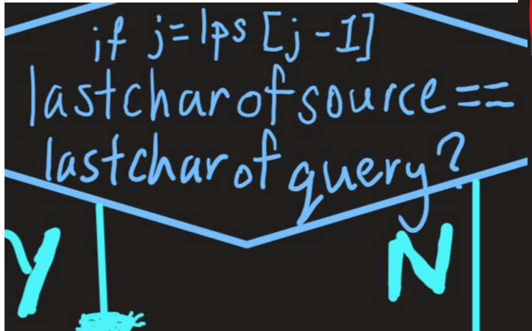

Start
i=0 ; j=0
N=len(source) ; M=len(query)
while i<N-M-1
N
End Search
Y
source[i] == query[j] ?
Y
j+=1 ; i+=1
N
j≠0?
Y
j=lps[j-1]
N
i+=1
j==M?
N
Y
Match found!
j=lps[j-1]

Normal KMP



Modified KMP





Time Complexity: Preprocessing

n is length of Source DNA, m is length of query pattern

- Bad Match Table: For all cases: approx. O(m)
- LPS: For all cases O(m)

Bad Match Table

A	C	G	T
1	3	3	3

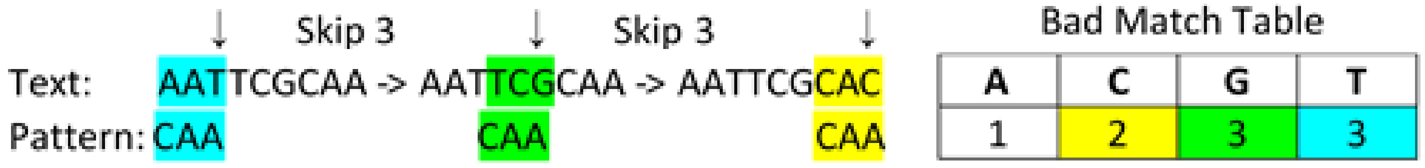
Pattern: ACTACG

lps arrary

A	C	T	A	C	G
0	0	0	1	2	0

Time Complexity:

- Best Case Scenario: O(n/m)
  - Occurs when if-else allows BMT⇒ causing algorithm to run like Boyer-Moore
  - Pointer i signifying pattern moving along source code increments by m each time, causing only n/m comparisons

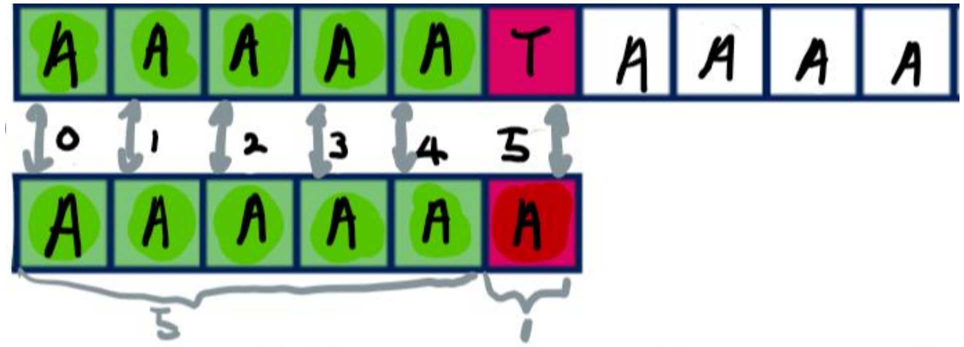


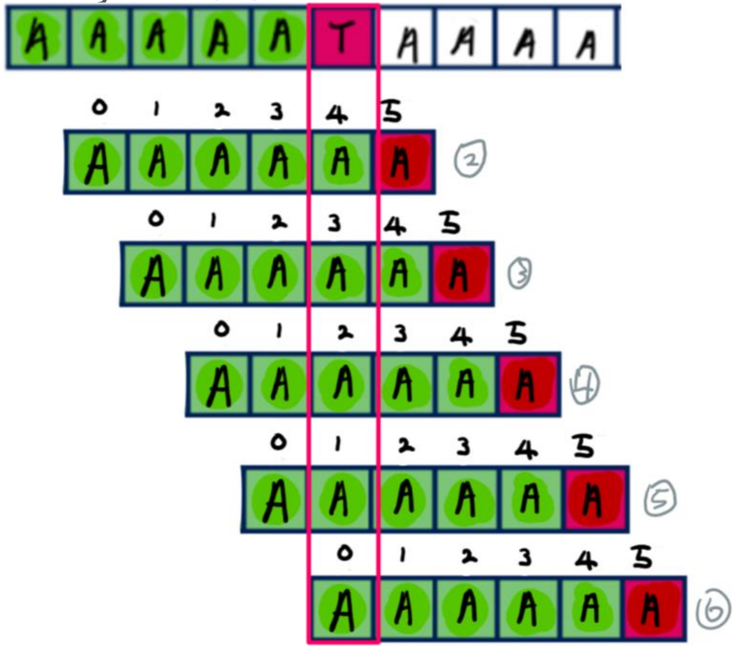
Bad Match Table

A	C	G	T
1	2	3	3

Time Complexity:

- Worst Case Scenario: O(n)
  - Text=AAAAATAAAAATAAAAATAAAA...
  - Pattern=AAAAAA , length of pattern = 6
  - Compares approx. 2n times: has complexity of O(n)





Time Complexity:

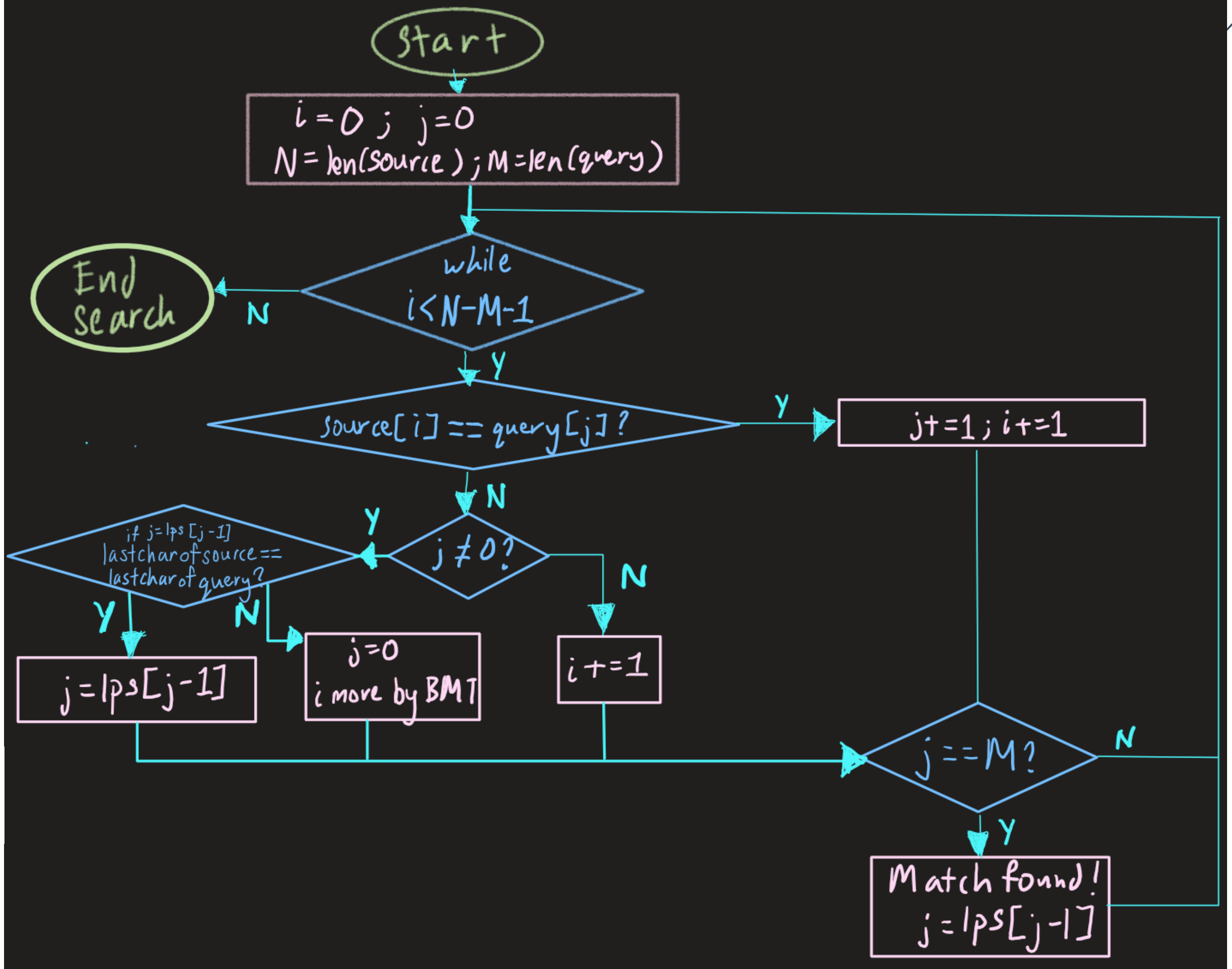
- Average Case Scenario: O(n)

Expected i shifts per iteration:

$$\frac{1}{4} + \frac{15+9m^2}{32m}$$

Expected number of comparisons:

$$n/(\frac{15+8m+9m^2}{32m}) = O(n)$$

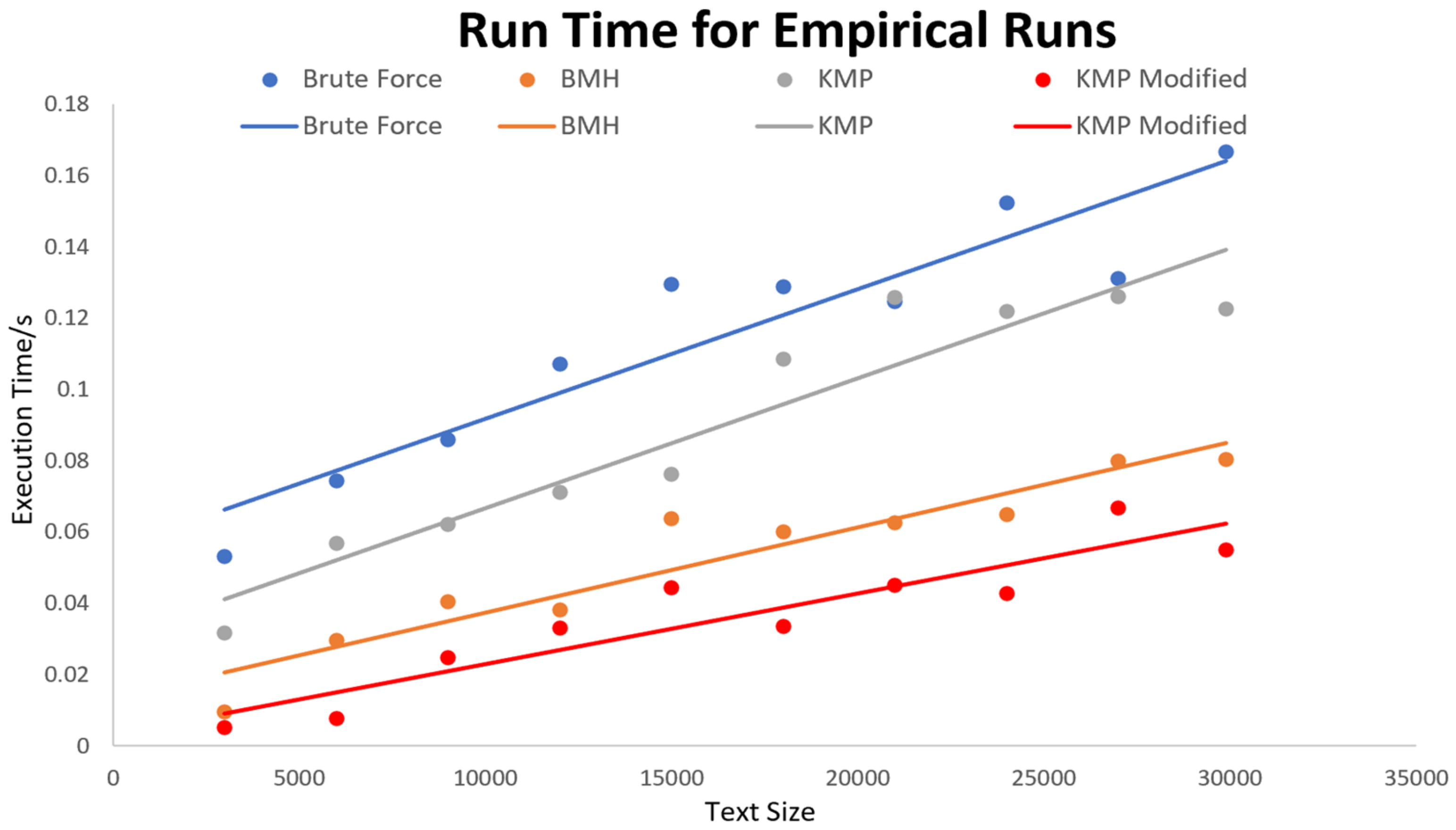


04

Conclusion

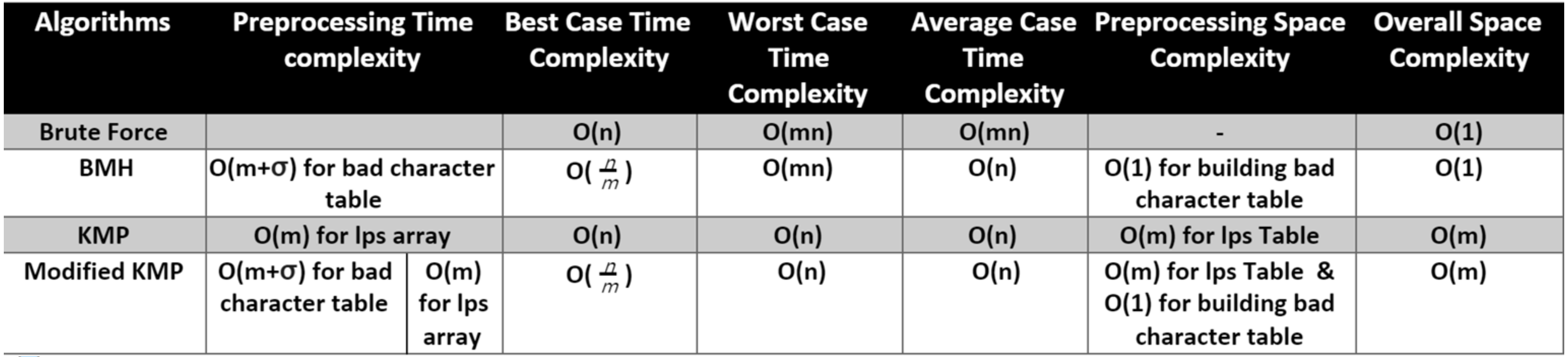
Comparison between algorithms

Comparison for empirical runs



Theoretical Complexity

* n: Length of text, m: Length of pattern, σ: Number of alphabets



Algorithms	Preprocessing Time complexity	Best Case Time Complexity	Worst Case Time Complexity	Average Case Time Complexity	Preprocessing Space Complexity	Overall Space Complexity
Brute Force		O(n)	O(mn)	O(mn)	-	O(1)
BMH	O(m+σ) for bad character table	O($\frac{n}{m}$)	O(mn)	O(n)	O(1) for building bad character table	O(1)
KMP	O(m) for lps array	O(n)	O(n)	O(n)	O(m) for lps Table	O(m)
Modified KMP	O(m+σ) for bad character table; O(m) for lps array	O($\frac{n}{m}$)	O(n)	O(n)	O(m) for lps Table & O(1) for building bad character table	O(m)

Thank You :)