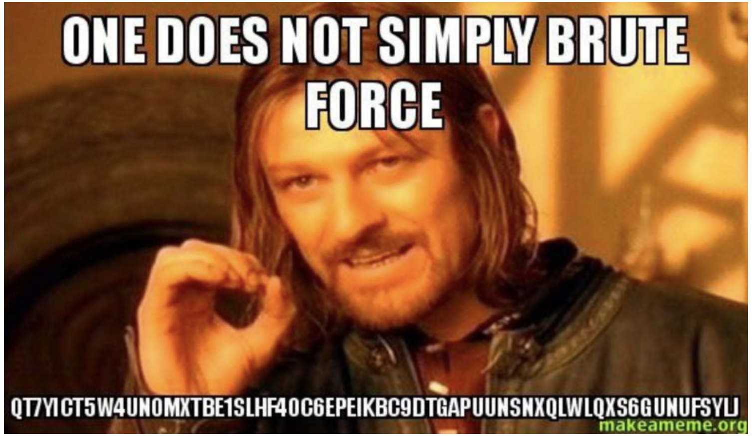
CS 1332R
WEEK 11

Introduction to Pattern Matching

Brute Force

Boyer-Moore

Knuth-Morris-Pratt



ANNOUNCEMENTS

- ❑

Introduction to Pattern Matching Algorithms

➔ **PROBLEM GOAL**: Finding a pattern (smaller string of characters) in a text (longer string of characters) - the same as your Command/Control F function.

Typically…

`n = text length` `m = pattern length`

`i = text index` `j = pattern index`

- **All Occurrences**: return a list of all indexes where the pattern occurs in the text
  - We must iterate through the entire text of length n.
- **Single Occurrence:** find the first index where the pattern occurs in the text
  - We stop at the first occurrence of the pattern.

Brute Force

INTUITION:

No optimizations, most basic search.

1. Line up index 0 of the pattern with index 0 of the text.
2. Compare each character of the pattern with each character in the text.
3. MISMATCH → shift pattern right by 1. Repeat step 1.

PSEUDOCODE:

```
procedure BruteForce(text, pattern):
  n is text's length, m is pattern's length
  for (i from 0 to n - m):
    j starts at 0
    while (j < m and pattern[j] matches text[i + j]):
      move j forward
    end while
    if (j == m):
      // match found at i
    end if
  end for
end procedure
```

Brute Force: Implementation

CODE OUTLINE:

`n = text length`

`m = pattern length`

`i = text index`

`j = pattern index`

→ We are comparing **`pattern[j]`** with **`text[i + j].`**

→ We increment **`pattern index (j)`** in the inner loop, while comparing.

→ We increment **`text index (i)`** and reset the **`pattern index (j)`** to 0 in the outer loop, after a mismatch.

PSEUDOCODE:

```
procedure BruteForce(text, pattern):
  n is text's length, m is pattern's length
  for (i from 0 to n - m):
    j starts at 0
    while (j < m and pattern[j] matches text[i + j]):
      move j forward
    end while
    if (j == m):
      // match found at i
    end if
  end for
end procedure
```

What initial check should we do before performing <u>any pattern matching algorithm</u>?

`m <= n`

In the outer loop above, why do stop at “n - m” instead of n?

After this index, the pattern extends past the end of the text.

Brute Force: Practice

Perform brute force to find a single occurrence of the pattern in the text:

text = mumunomummy

pattern = mummy

`n = 11, m = 5`

	0	1	2	3	4	5	6	7	8	9	10
text	m	u	m	u	n	o	m	u	m	m	y
pattern	m	u	m	m	y						

= mismatch

= match

Brute Force: Practice

	0	1	2	3	4	5	6	7	8	9	10
text	m	u	m	u	n	o	m	u	m	m	y
pattern	m	u	m	m	y						
		m	u	m	m	y					
			m	u	m	m	y				
1 iteration of the outer loop →				m	u	m	m	y			
					m	u	m	m	y		
						m	u	m	m	y	
							m	u	m	m	y

Answer: 6

How many comparisons did we make in this example?

16

On an exam, you must circle *all* comparisons made throughout the algorithm - the red AND green boxes.

Brute Force: Efficiencies

Best	Average	Worst
O(m) or O(n)	*O(mn)*	*O(mn)*

Single: O(m) - occurrence at i = 0

text = baaacdegfbaaa

pattern = baaa

All: O(n) - first letter of pattern is not in the text

text = baaabaaa

pattern = dee

text = aaaaaaaaaac

pattern = aac

Boyer-Moore

INTUITION: If a character in the text is not present in the pattern, we can move our pattern completely past this character in the text.

Part 1: Last Occurrence Table

- a map of each character in the pattern to the last index it occurred in the pattern
- O(m) - iterate through the pattern, updating the map as you go

Pattern: Queue

Last Occurrence Table:

Q	U	E	*
0	3	4	-1

```
lot.getOrDefault(key, -1)
```

Boyer-Moore: Implementation

INTUITION: If a character in the text is not present in the pattern, we can move our pattern completely past this character in the text.

Part 2: Algorithm

1. Start `i = 0`, `j = m - 1`. **We compare from right to left within the pattern.** Decrement j.
2. If mismatch…
   a. `lot.get(text[i + j]) == -1` → shift pattern completely past the text,
   b. `lot.get(text[i + j]) < j` → shift pattern *forward* to last occurrence of the text character in the pattern
   c. `lot.get(text[i + j]) > j` → shift pattern over by 1 (increment `i`)

```
procedure BoyerMoore(text, pattern):
  lastTable is pattern's last occurrence table
  m is pattern's length, n is text's length
  i starts at 0
  while (i <= n - m):
    j starts at m - 1
    while (j >= 0 and text[i + j] matches pattern[j]):
      decrement j
    end while
    if (j == -1):
      // match found at i
      move i forward
    else:
      shift is the lastTable index for text[i + j]
      if (shift < j):
        add j - shift to i
      else:
        move i forward
      end if
    end if
  end while
end procedure
```

match

mismatch

Boyer-Moore: Practice

Perform Boyer-Moore to find *all occurrences* of the pattern in the text:

text = quequeuedequeue

pattern = queue

`n = 15, m = 5`

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
text	q	u	e	q	u	e	u	e	d	e	q	u	e	u	e
pattern	q	u	e	u	e										

Q	U	E	*
0	3	4	-1

Boyer-Moore: Practice

Iteration	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	q	u	e	q	u	e	u	e	d	e	q	u	e	u	e
1	q	u	e	u	e										
2		q	u	e	u	e									
3															
4															
5															
6															
7															

Answer:

Q	U	E	*
0	3	4	-1

Boyer-Moore: Practice

Iteration	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	q	u	e	q	u	e	u	e	d	e	q	u	e	u	e
1	q	u	e	u	e										
2		q	u	e	u	e									
3				q	u	e	u	e							
4															
5															
6															
7															

Answer:

Q	U	E	*
0	3	4	-1

Boyer-Moore: Practice

Iteration	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	q	u	e	q	u	e	u	e	d	e	q	u	e	u	e
1	q	u	e	u	e										
2		q	u	e	u	e									
3				q	u	e	u	e							
4					q	u	e	u	e						
5															
6															
7															

Answer: 3

Q	U	E	*
0	3	4	-1

Boyer-Moore: Practice

Iteration	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	q	u	e	q	u	e	u	e	d	e	q	u	e	u	e
1	q	u	e	u	e										
2		q	u	e	u	e									
3				q	u	e	u	e							
4					q	u	e	u	e						
5										q	u	e	u	e	
6															
7															

Answer: 3

Q	U	E	*
0	3	4	-1

Boyer-Moore: Practice

Iteration	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	q	u	e	q	u	e	u	e	d	e	q	u	e	u	e
1	q	u	e	u	e										
2		q	u	e	u	e									
3				q	u	e	u	e							
4					q	u	e	u	e						
5										q	u	e	u	e	
6											q	u	e	u	e
7															

Answer: 3

Q	U	E	*
0	3	4	-1

Boyer-Moore: Practice

Iteration	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	q	u	e	q	u	e	u	e	d	e	q	u	e	u	e
1	q	u	e	u	e										
2		q	u	e	u	e									
3				q	u	e	u	e							
4					q	u	e	u	e						
5										q	u	e	u	e	
6											q	u	e	u	e
7															

Answer: 3, 10

When do we see the greatest shift?
What does this hint to us about the best-case scenario of Boyer-Moore?

When the text character is not in the pattern, we shift by m. The best case is when we always shift by m.

Boyer Moore: Efficiencies

Best	Average	Worst
Single: O(m) *All: O(n/m + m)*	*very text/pattern dependent*	*O(m + mn) → O(mn)*

Single: O(m) - occurrence at i = 0

text = baaacdegfbaaa

pattern = baaa

All: O(n/m + m) - text character compared with last character of pattern is never in the pattern

text = aacaacaacaac

pattern = aab

If we constantly shift by 1, Boyer-Moore kind of degenerates into Brute Force.

Case 1: always mismatch on the last character

text = aaaaaaaaaa

pattern = caa

Case 2: always match

text = aaaaaaaaaa

pattern = aaa

INTUITION: Boyer Moore works best when the pattern and text have low overlap between their characters, making larger shifts of the pattern more likely.

Knuth-Morris-Pratt (KMP)

INTUITION: Use matching prefixes and suffixes within the pattern (characters that are repeated in the beginning and end of the pattern) to optimize shifting and reduce the number of comparisons we make.

Part 1: Failure Table

- an array of length m
- failureTable[n] contains the ***length of the longest prefix that is also a suffix of the pattern up to that point***

Pattern: abaababac

Failure Table:

Index:	0	1	2	3	4	5	6	7	8
Char:	a	b	a	a	b	a	b	a	c
Value:	0	0	1	1	2	3	2	3	0

KMP: Failure Table Implementation

INTUITION: Use matching prefixes and suffixes within the pattern (characters that are repeated in the beginning and end of the pattern) to optimize shifting and reduce the number of comparisons we make.

```
procedure BuildFailureTable(pattern):
  m is pattern's length
  failureTable is an array of length m
  i starts at 0, j starts at 1
  set the first failureTable value to 0
  while (j < m):
    if (pattern[i] matches pattern[j]):
      set failureTable at index j to i + 1
      move i and j forward
    else:
      if (i is 0):
        set failureTable at index j to 0
        move j forward
      else:
        move i to previous value in failureTable
      end if
    end if
  end while
  return failureTable
end procedure
```

Pattern: abaababac

Failure Table:

Index:	0	1	2	3	4	5	6	7	8
Char:	a	b	a	a	b	a	b	a	c
Value:	0	0	1	1	2	3	2	3	0

O(m)

KMP: Implementation

Part 2: Algorithm

1. Start `i = 0`, `j = 0`. **We compare left to right within the pattern.** Increment j while matching characters.
2. If…
   a. `j == 0` → `i++`
   b. `j != 0` →
      i. `j = failureTable[j - 1]`
      ii. `i += j - shift`

```
procedure KMP(text, pattern):
  failureTable is pattern's failure table
  m is pattern's length, n is text's length
  i and j start at 0
  while (i <= n - m):
    while (j < m and text[i+j] matches pattern[j]):
      move j forward
    end while
    if (j is 0):
      move i forward
    else:
      if (j is m):
        // match found at i
      end if
      shift is the failureTable value at j - 1
      move i forward by j - shift
      set j to shift
    end if
  end while
end procedure
```

KMP: Practice

Create the failure table for the pattern. Then perform KMP to find all occurrences.

text = aabababbabababac

pattern = ababac

`n = 14, m = 6`

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
text	a	a	b	a	b	a	b	b	a	b	a	b	a	c
pattern	a	b	a	b	a	c								

HINT:

failure table	0	0	1	2	3	0

KMP: Practice

failure table

0	0	1	2	3	0

ITERATION	0	1	2	3	4	5	6	7	8	9	10	11	12	13
	a	a	b	a	b	a	b	b	a	b	a	b	a	c
1	a	b	a	b	a	c								
2		a	b	a	b	a	c							
3														
4														
5														
6														
7														

Answer:

failure table

0	0	1	2	3	0

KMP: Practice

ITERATION	0	1	2	3	4	5	6	7	8	9	10	11	12	13
	a	a	b	a	b	a	b	b	a	b	a	b	a	c
1	a	b	a	b	a	c								
2		a	b	a	b	a	c							
3				a	b	a	b	a	c					
4														
5														
6														
7														

Answer:

failure table

0	0	1	2	3	0

KMP: Practice

ITERATION	0	1	2	3	4	5	6	7	8	9	10	11	12	13
	a	a	b	a	b	a	b	b	a	b	a	b	a	c
1	a	b	a	b	a	c								
2		a	b	a	b	a	c							
3				a	b	a	b	a	c					
4						a	b	a	b	a	c			
5														
6														
7														

Answer:

failure table

0	0	1	2	3	0

KMP: Practice

ITERATION	0	1	2	3	4	5	6	7	8	9	10	11	12	13
	a	a	b	a	b	a	b	b	a	b	a	b	a	c
1	a	b	a	b	a	c								
2		a	b	a	b	a	c							
3				a	b	a	b	a	c					
4						a	b	a	b	a	c			
5								a	b	a	b	a	c	
6														
7														

Answer:

failure table

0	0	1	2	3	0

KMP: Practice

ITERATION	0	1	2	3	4	5	6	7	8	9	10	11	12	13
	a	a	b	a	b	a	b	b	a	b	a	b	a	c
1	a	b	a	b	a	c								
2		a	b	a	b	a	c							
3				a	b	a	b	a	c					
4						a	b	a	b	a	c			
5								a	b	a	b	a	c	
6									a	b	a	b	a	c
7														

Answer:

failure table

0	0	1	2	3	0

KMP: Practice

ITERATION	0	1	2	3	4	5	6	7	8	9	10	11	12	13
	a	a	b	a	b	a	b	b	a	b	a	b	a	c
1	a	b	a	b	a	c								
2		a	b	a	b	a	c							
3				a	b	a	b	a	c					
4						a	b	a	b	a	c			
5								a	b	a	b	a	c	
6									a	b	a	b	a	c
7														

Answer: 8

Why do we not have to compare the characters in the yellow blocks?

The failure table told us that the previous last couple of characters are also a prefix of the pattern. Therefore, we already know these characters match with the text.

KMP: Efficiencies

Best	Average	Worst
O(m) or O(m + n)	*O(m + n)*	*O(m + n)*

Single: O(m)

text = baaacdegfbaaa

pattern = baaa

All: O(m + n)

Case 1: shift by m every time

text = aabaabaabaab

pattern = aab

Case 2: shift by 1 every time - first m - 2 characters are not compared

text = aaaaaaaaaaaa

pattern = aaa

INTUITION: KMP works best for a small alphabet, patterns are more likely to occur within smaller alphabets.

Pattern Matching: Efficiencies

Brute Force

Scenario	Best	Best Ex	Worst	Worst Ex
No Occurrences	$O(n)$	P: baa T: aaaaaaa	$O(mn)$	P: aab T: aaaaaaa
Single Occurrences	$O(m)$	P:aaa T: aaaaaaa	$O(mn)$	P: aab T: aaaaaaab
All Occurrences	$O(n)$	P: baa T: baaaaabaa	$O(mn)$	P: aaab T: aaaaaaab

Boyer Moore

Scenario	Best	Worst
LOT (preprocess)	$O(m)$	$O(m)$
No Occurrences	$O(m + n/m)$	$O(mn)$
Single Occurrence	$O(m)$	$O(mn)$
All Occurrences	$O(m + n/m)$	$O(mn)$

KMP

Scenario	Best	Worst
No Occurrences	$O(m + n)$	$O(m + n)$
Single Occurrence	$O(m)$	$O(m + n)$
All Occurrences	$O(m + n)$	$O(m + n)$

Pattern Matching: Practice

1. Best case of Brute Force string searching with text of length n and pattern length of m when trying to find all occurrences of the pattern? **O(n)**
2. Worst case of Brute Force string searching with text of length n and pattern length of m? **O(mn)**
3. If I know my alphabet only has 5 characters, should I use KMP or BM? **KMP**
4. If I know my alphabet can be any alphanumeric character, should I use KMP or BM? **BM**
5. How many times would we shift the pattern if no character in the text exists in the pattern for BM, as a function of n and m? **n/m**

LEETCODE PROBLEMS

[187. Repeated DNA Sequences]

[229. Majority Element II]

[1392. Longest Happy Prefix]

Any questions?

Name
Office Hours
Contact

Name
Office Hours
Contact

Let us know if there is anything specific you want out of recitation!