Chapter 15: Stringing Along

Panos Louridas

Athens University of Economics and Business
Real World Algorithms
A Beginners Guide
The MIT Press

Outline

General

- Searching through text is a very common task.
- It happens every time we search for something in our browser, or in a PDF document, or inside a document in our word processor, etc.
- All these are instances of the same underlying operation: *string matching* or *string search*.

String Matching beyond Text

- Many times, our search is not inside a real text.
- However, the same principles apply to any situation where we try to find something inside something else, where both the search item and the search area are sequences of symbols from the same alphabet.

The Genetic Code

- In biology, the *genetic code* is the set of rules by which DNA and RNA encode proteins.
- The DNA consists of four bases: adenine (A), guanine (G), cytosine (C), and thymine (T).
- A triplet of bases, called a *codon*, encodes one particular amino acid. A sequence of codons encodes a particular protein; such a codon sequence is a *gene*.

Table of the DNA Genetic Code

(1)	T		C		A		G		(3)
				(2)					
T	TTT	Phenylalanine	TCT	Serine	TAT	Tyrosine	TGT	Cysteine	T
	TTC		TCC		TAC		TGC		C
	TTA	Leucine	TCA		TAA	Stop	TGA	Stop	A
	TTG		TCG		TAG	Stop	TGG	Tryptophan	G
C	CTT		CCT	Proline	CAT	Histidine	CGT	Arginine	T
	CTC		CCC		CAC		CGT		C
	CTA		CCA		CAA	Glutamine	CGA		A
	CTG		CCG		CAG		CGG		G
A	ATT	Isoleucine	ACT	Threonine	AAT	Asparagine	AGT	Serine	T
	ATC		ACC		AAC		AGC		C
	ATA		ACA		AAA	Lysine	AGA	Arginine	A
	ATG	Methionine / Start	AGC		AAG		AGG		G
G	GTT	Valine	GCT	Alanine	GAT	Aspartic acid	GGT	Glycine	T
	GTC		GCC		GAC		GGC		C
	GTA		GCA		GAA	Glutamic acid	GGA		A
	GTG		GCG		GAG		GGG		G

String Matching in Genetic Material

- When we try to find a DNA sequence we use string matching techniques.
- The full human genome is estimated to contain about 3.2 million bases.
- Therefore we must make sure that string matching is efficient.

Other String Matching Applications

- Electronic surveillance.
- Computer forensics.
- Intrusion detection, intrusion prevention.
- Spam detection.
- Web scraping, screen scraping.

Kinds of String Matching

Different string matching applications have different requirements:

- In *exact matching* we want to find precisely a string.
- In *approximate matching* we want to find variants of it.
- Also, the size of the alphabet is important in search techniques.

Definitions

Definition

We will call *pattern* the string we are trying to and and *text* the string in which we are trying to find the pattern.

Note that the text can be any kind of string, not just humanly readable text.

Outline

Brute Force String Matching

- The simplest string matching method does not use any kind of intelligence.
- In *brute force* string matching we start at the beginning of the text and check for a match letter by letter.
- If we fail, we advance one letter in the text and try to match again, and so on.

Brute Force String Matching Algorithm

Algorithm: Brute force string search.

`BruteForceStringSearch`$(p, t) \rightarrow q$

Input: p, a pattern
t, a text

Output: q, a queue containing the indices of t where p is found; if p is not found the queue is empty

```
1  q ← CreateQueue()
2  m ← |p|
3  n ← |t|
4  for i ← 0 to n − m do
5      j ← 0
6      while j < m and p[j] = t[i + j] do
7          j ← j + 1
8      if j = m then
9          Enqueue(q, i)
10 return q
```

Brute Force Example

i	j	B	A	D	B	A	R	B	A	R	D
0	2	B	A	R	D						
1	0		B	A	R	D					
2	0			B	A	R	D				
3	3				B	A	R	D			
4	0					B	A	R	D		
5	0						B	A	R	D	
6	4							B	A	R	D

Worst Case Brute Force

i	j	0	0	0	0	0	0	0	0	0	1
0	3	0	0	0	1						
1	3		0	0	0	1					
2	3			0	0	0	1				
3	3				0	0	0	1			
4	3					0	0	0	1		
5	3						0	0	0	1	
6	4							0	0	0	1

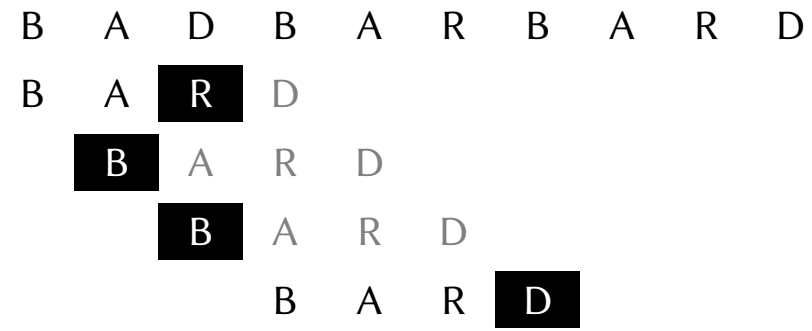
Brute Force Complexity

- The outer loop is executed $n - m$ times.
- The worst case scenario is to check all the characters of p and to find a mismatch only in the last character.
- If this happens, we need m iterations of the internal loop.
- So in total we have $O(m(n - m))$ complexity.
- Because usually n is much bigger than m, we can simplify the above to $O(mn)$.

Outline

Improvement Opportunities (1)

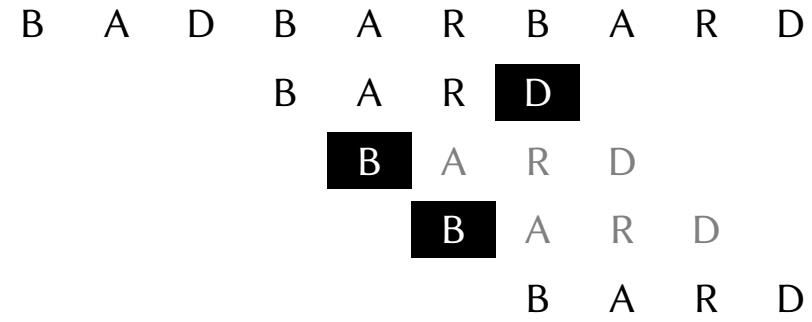
If we go back to our last example, we can see that there are opportunities for improvement.



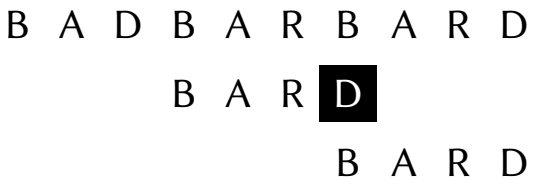
The second and the third try are doomed, and we know that from the first try! We should start directly three characters to the right:

Improvement Opportunities (2)

But the same thing happens again:



As we already know that the text at point is BARB, we can jump directly three characters to the right:



Improvement Opportunities (3)

Let's see another example:

B A B A B A B C A B C

A B A B C

The first character is a mismatch, so we go one character to the right:

B A B A B A B C A B C

A B A B C

Improvement Opportunities (4)

We failed at the fourth character, so we can assume that we can start again after moving four characters to the right:

B A B A B A B C A B C

A B A B C

However, this would be *wrong*, because we would miss the match we would get by moving two characters to the right:

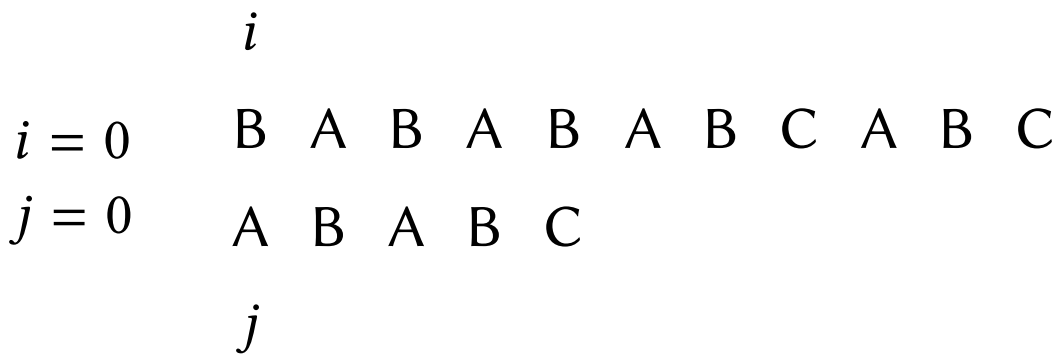
B A B A B A B C A B C

A B A B C

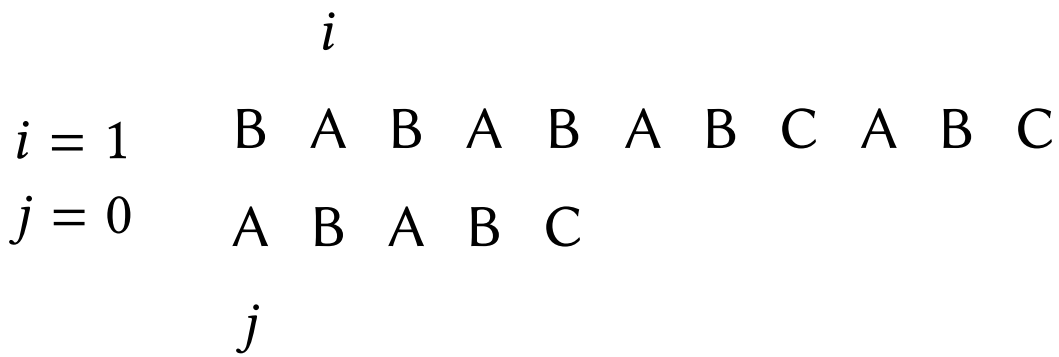
Basic Idea of the Knuth-Morris-Pratt Algorithm

- We proceed character by character in the text.
- Suppose that we are at position i of the text and we have matched j from the pattern.
- We increase i to $i + 1$.
- We check whether the $(j + 1)$th character of the text matches with the $(i + 1)$th character of the pattern.
- If yes, we increase i and j.
- If no, we try to find how many characters we can match at position $i + 1$, and we update j accordingly.

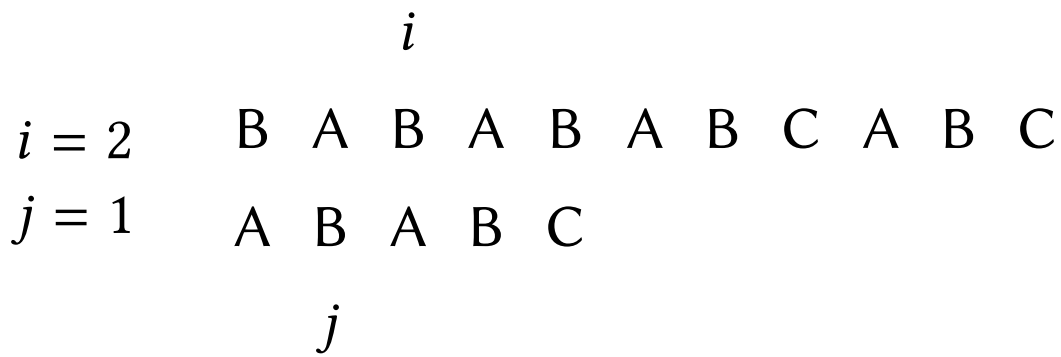
Knuth-Morris-Pratt Example (1)



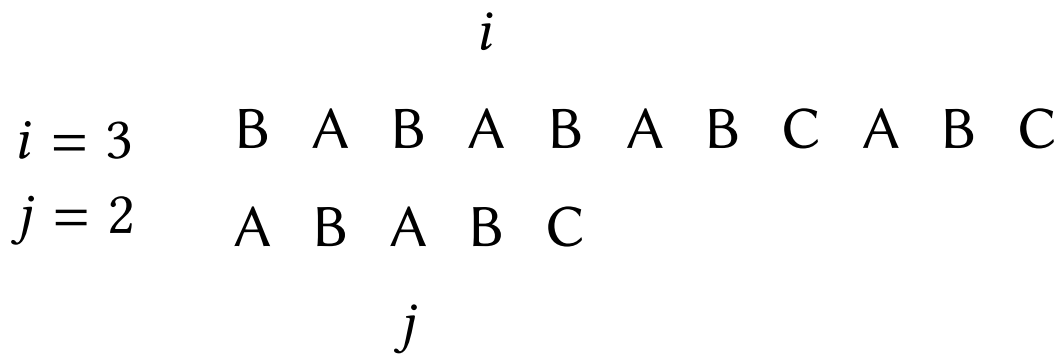
Knuth-Morris-Pratt Example (2)



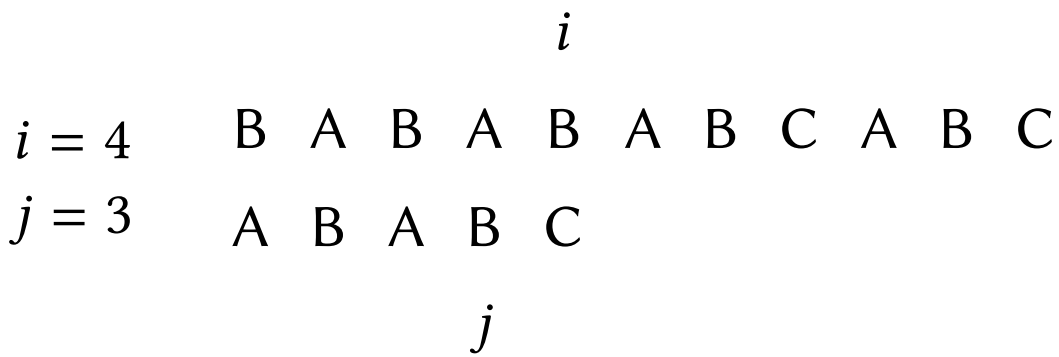
Knuth-Morris-Pratt Example (3)



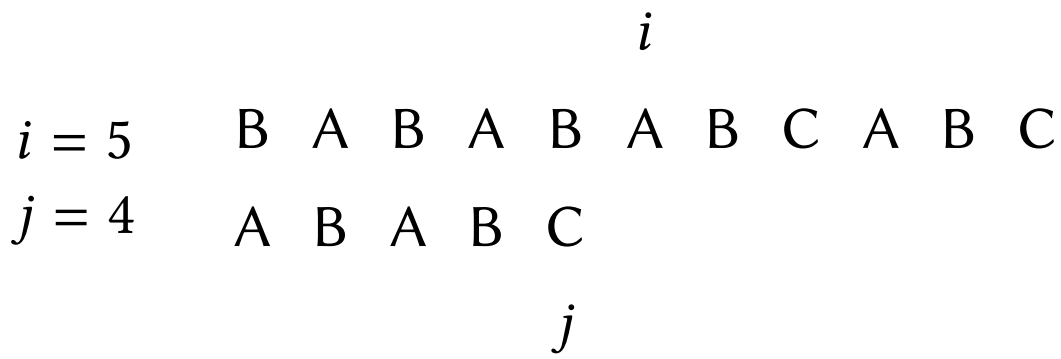
Knuth-Morris-Pratt Example (4)



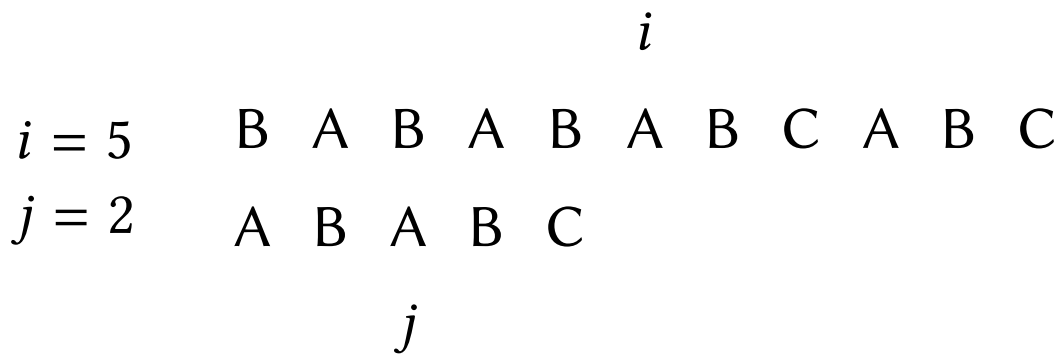
Knuth-Morris-Pratt Example (5)



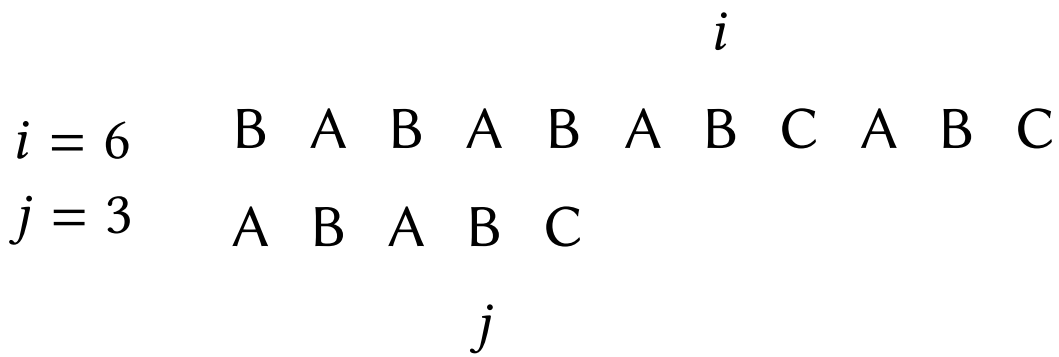
Knuth-Morris-Pratt Example (6)



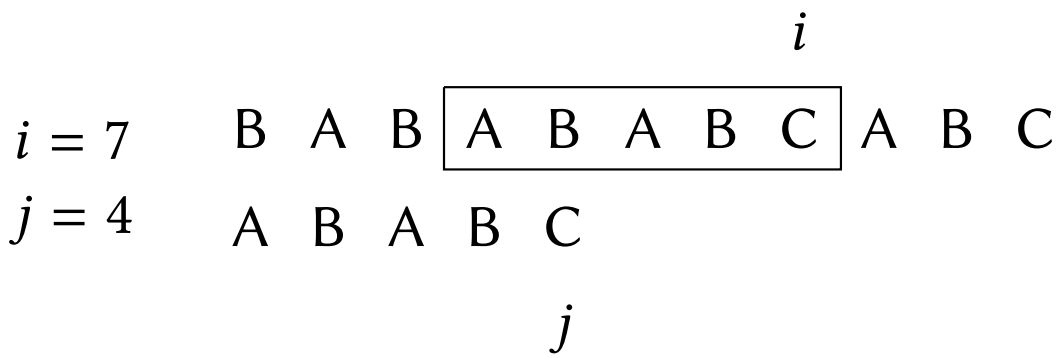
Knuth-Morris-Pratt Example (7)



Knuth-Morris-Pratt Example (8)

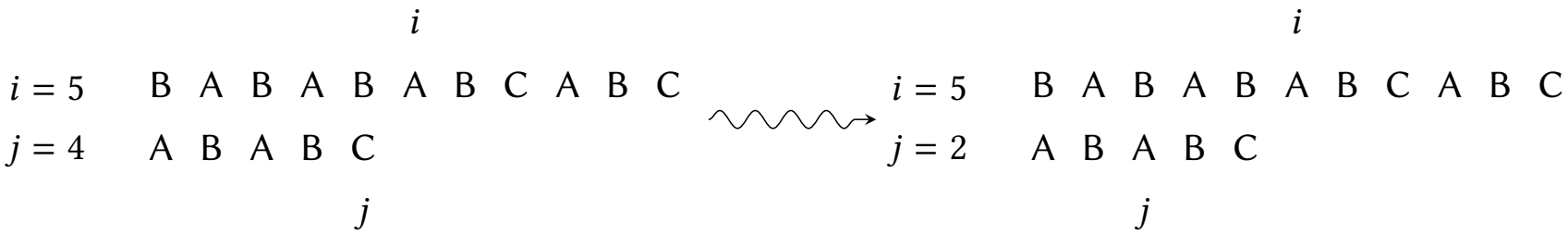


Knuth-Morris-Pratt Example (9)

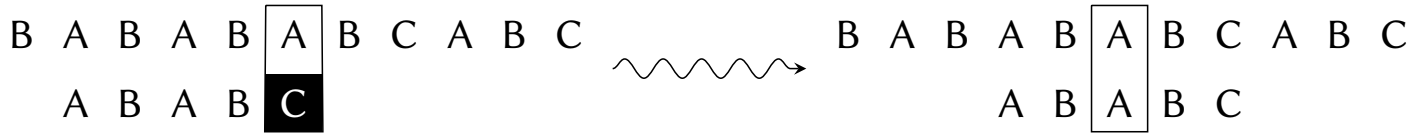


Shifting the Pattern

Alternatively, we can view the change in j as shifting the pattern to the right. That is, this:



is the same with this:



Terminology (1)

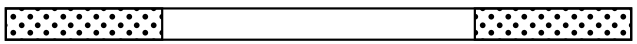
- A part of a string is called a *substring*.
- A part of a string at the start of the string is called a *prefix*.
- A, AB, ABA, ... are prefixes of string ABAXYZABA.
- The empty substring is considered to be the prefix of every string.
- A string is a prefix of itself.
- A *proper prefix* is a prefix that is not the whole string.
- Alternatively, we may also require that a proper prefix is not equal to the empty substring.
- We will be dealing with non-empty prefixes.

Terminology (2)

- A part of a string at the end of a string is called a *suffix*.
- A, BA, ABA, ... are suffixes of ABAXYZABA.
- The empty substring is considered to be a suffix of every string.
- A string is a suffix of itself.
- A *proper suffix* is a suffix that is not the whole string.
- Alternatively, we may also require that a proper suffix is not equal to the empty substring.
- We will be dealing with non-empty suffixes.

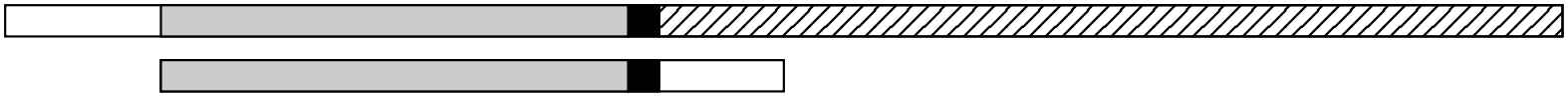
Terminology (3)

- A *border* is a proper prefix that is also a proper suffix.
- ABA is a border of ABAXYZABA.
- The *maximum border* of a string is the border with the maximum length.
- In ABAXYZABA substrings A and ABA are borders and ABA is the maximum border.
- If there is no maximum border, we say that its length is zero.

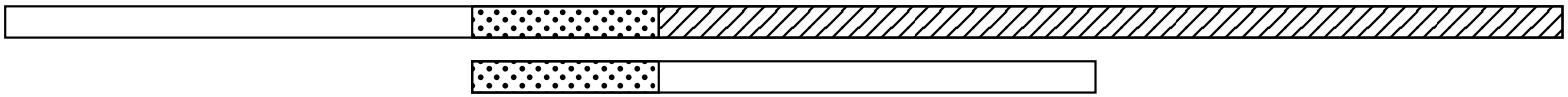


Borders and Matching

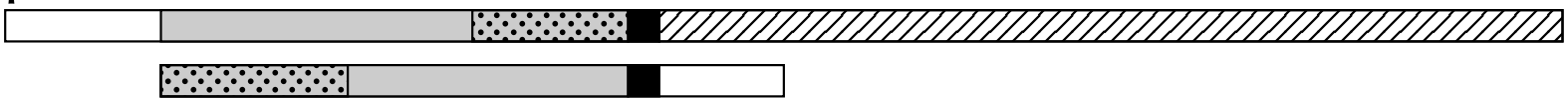
- Suppose that we have the following mismatch:



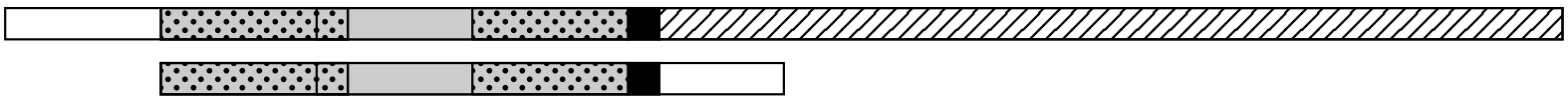
- Moreover, suppose that by shifting the pattern to the right, we can match some characters up to and including the mismatch:



- And now suppose that we bring back the pattern to its previous position:



- That means we must have a border!

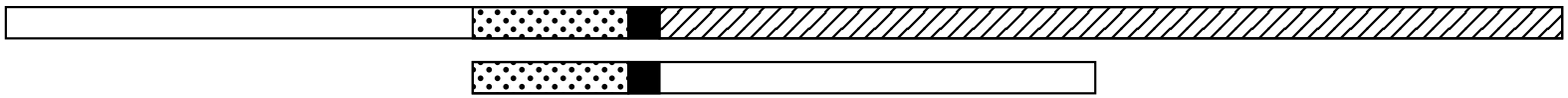


Borders and Matching

- It follows that if we find a mismatch, we may be able to reuse part of the matched pattern, provided it is a border.
- It may come out that we still don't get a match, but it's worth trying.
- Moreover, we should start with the longest border and then proceed with smaller ones, so that we don't miss a possible match.
- That is because *longer* borders correspond to *shorter* shifts, while *longer* shifts correspond to *shorter* borders.

Borders and Matching Example

Compare this:

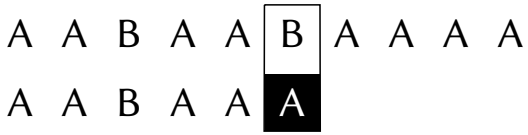


with this:

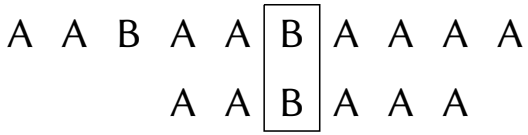
The second example has a longer border and therefore a shorter shift to the right.

Another Borders and Matching Example

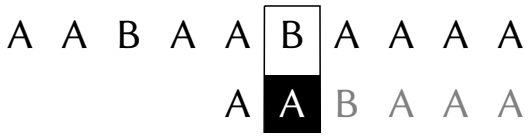
- Suppose we are searching for AABAAA in AABAABAAAA. We find that we can match the first five characters:



- The matched prefix is AABAA, which has two borders, AA and A.
- We must start with AA, which corresponds to a shift of three positions and a full match.



- If instead of that we had tried with A, which corresponds to a shift of four positions, we would miss the match:



Border Arrays

Border array for ABCABCACAB:

j	0	1	2	3	4	5	6	7	8	9	10
		A	B	C	A	B	C	A	C	A	B
$b[j]$	0	0	0	0	1	2	3	4	0	1	2

Border array for AABAAA:

j	0	1	2	3	4	5	6
		A	A	B	A	A	A
$b[j]$	0	0	1	0	1	2	2

The Knuth-Morris-Pratt Algorithm

Algorithm: Knuth-Morris-Pratt.

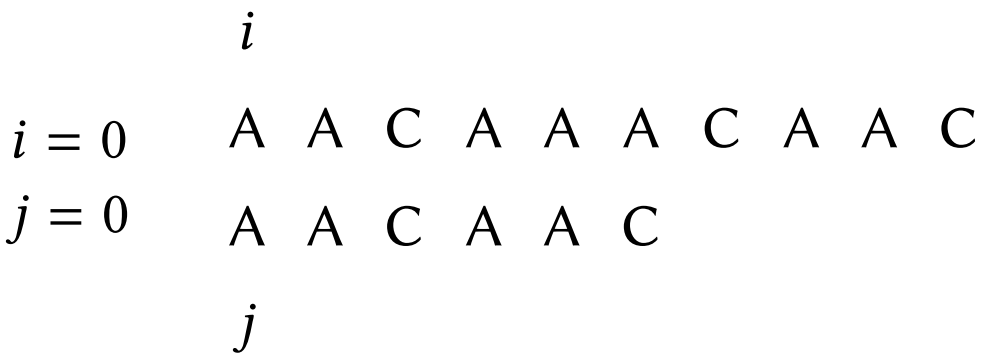
KnuthMorrisPratt$(p, t) \rightarrow q$

Input: p, a pattern
t, a text

Output: q, a queue containing the indices of t where p is found; if p is not found the queue is empty

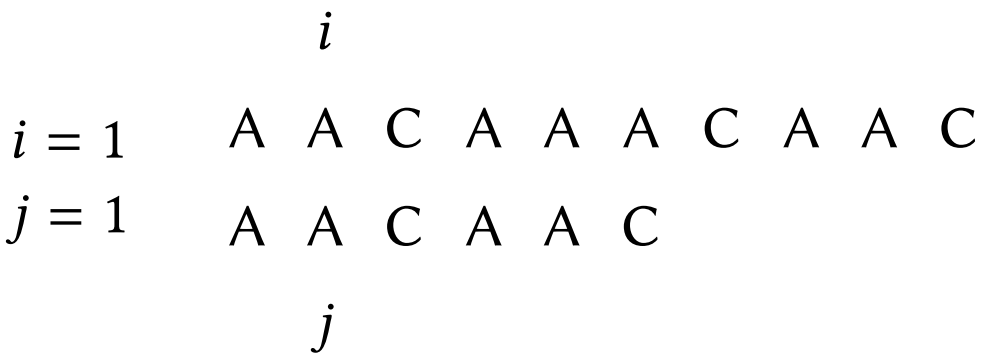
```
1  q ← CreateQueue()
2  m ← |p|
3  n ← |t|
4  b ← FindBorders(p)
5  j ← 0
6  for i ← 0 to n do
7      while j > 0 and p[j] ≠ t[i] do
8          j ← b[j]
9      if p[j] = t[i] then
10         j ← j + 1
11     if j = m then
12         Enqueue(q, i − j + 1)
13         j ← b[j]
14 return q
```

Knuth-Morris-Pratt Example (1)



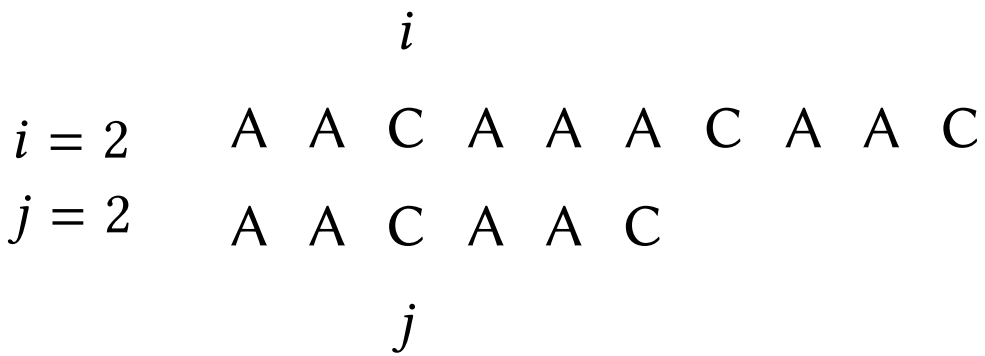
j	0	1	2	3	4	5	6
		A	A	C	A	A	C
$b[j]$	0	0	1	0	1	2	3

Knuth-Morris-Pratt Example (2)



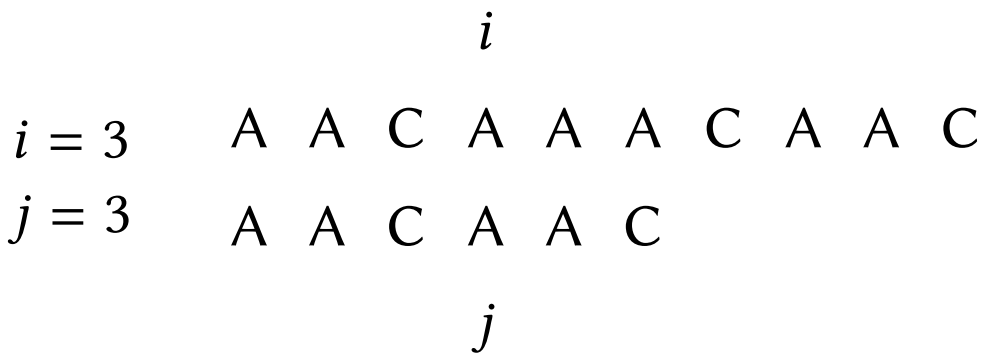
j	0	1	2	3	4	5	6
		A	A	C	A	A	C
$b[j]$	0	0	1	0	1	2	3

Knuth-Morris-Pratt Example (3)



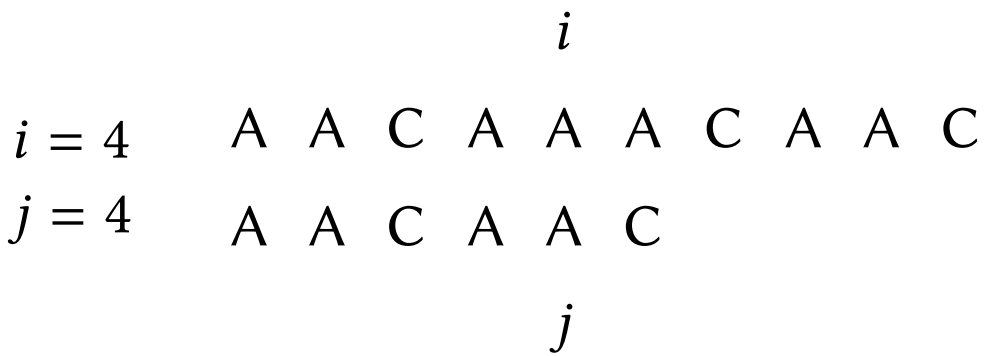
j	0	1	2	3	4	5	6
		A	A	C	A	A	C
$b[j]$	0	0	1	0	1	2	3

Knuth-Morris-Pratt Example (4)



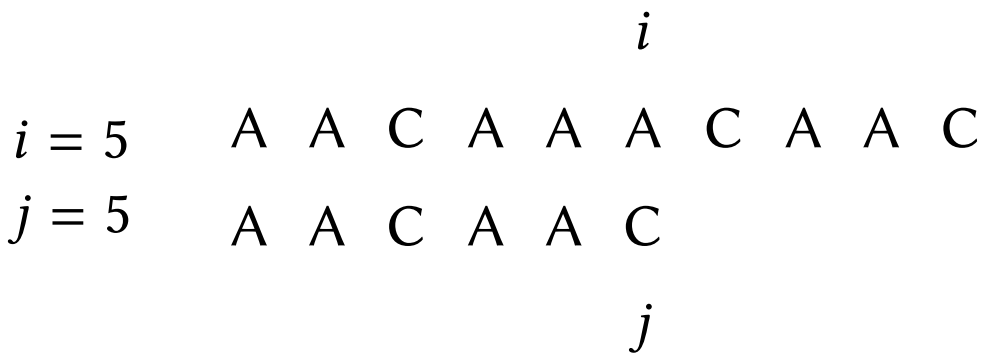
j	0	1	2	3	4	5	6
		A	A	C	A	A	C
$b[j]$	0	0	1	0	1	2	3

Knuth-Morris-Pratt Example (5)



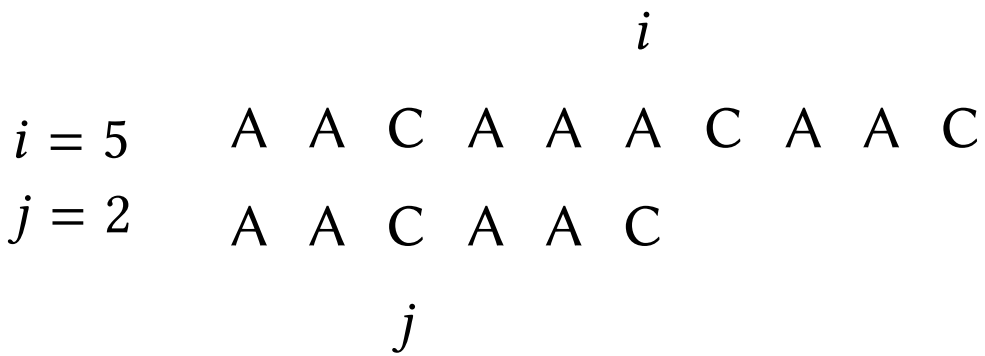
j	0	1	2	3	4	5	6
		A	A	C	A	A	C
$b[j]$	0	0	1	0	1	2	3

Knuth-Morris-Pratt Example (6)



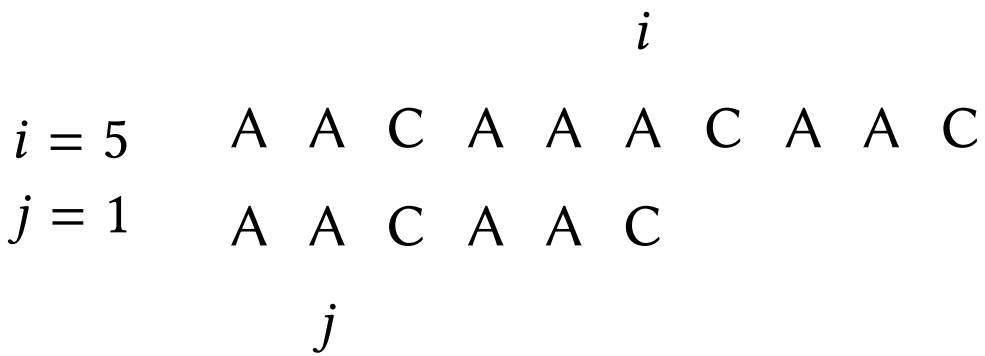
j	0	1	2	3	4	5	6
		A	A	C	A	A	C
$b[j]$	0	0	1	0	1	2	3

Knuth-Morris-Pratt Example (7)



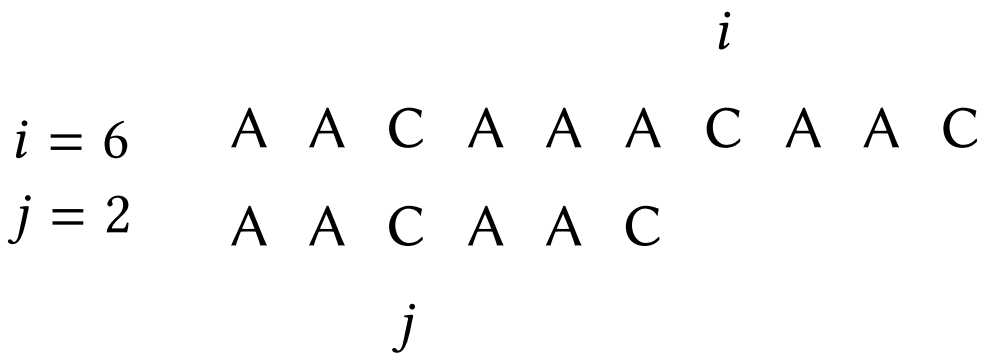
j	0	1	2	3	4	5	6
		A	A	C	A	A	C
$b[j]$	0	0	1	0	1	2	3

Knuth-Morris-Pratt Example (8)



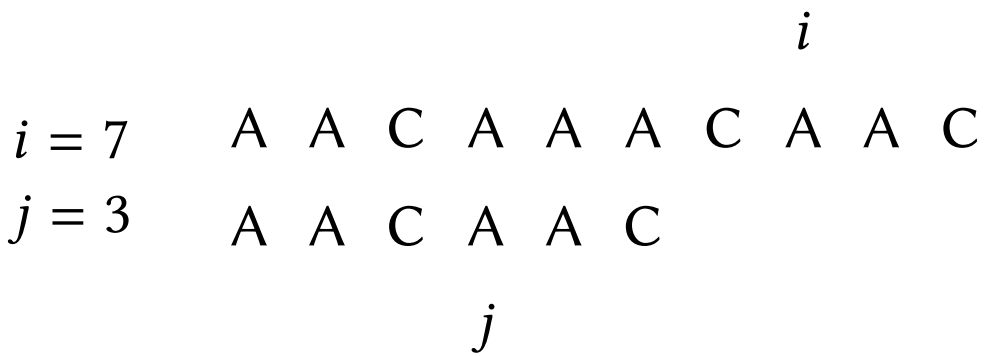
j	0	1	2	3	4	5	6
		A	A	C	A	A	C
$b[j]$	0	0	1	0	1	2	3

Knuth-Morris-Pratt Example (9)



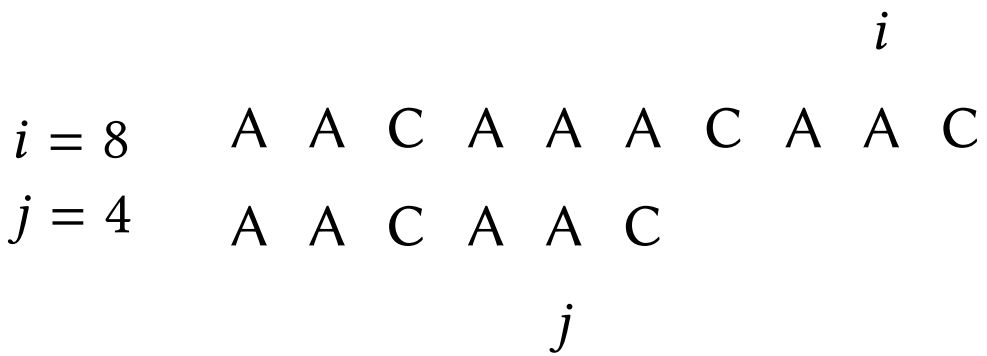
j	0	1	2	3	4	5	6
		A	A	C	A	A	C
$b[j]$	0	0	1	0	1	2	3

Knuth-Morris-Pratt Example (10)



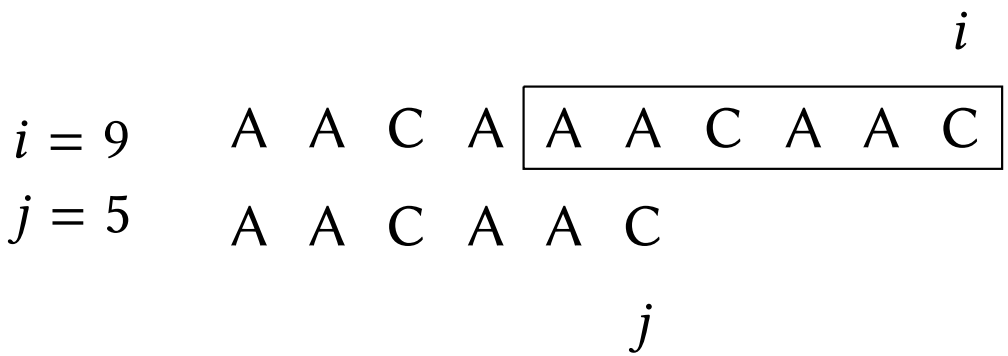
j	0	1	2	3	4	5	6
		A	A	C	A	A	C
$b[j]$	0	0	1	0	1	2	3

Knuth-Morris-Pratt Example (11)



j	0	1	2	3	4	5	6
		A	A	C	A	A	C
$b[j]$	0	0	1	0	1	2	3

Knuth-Morris-Pratt Example (12)



j	0	1	2	3	4	5	6
		A	A	C	A	A	C
$b[j]$	0	0	1	0	1	2	3

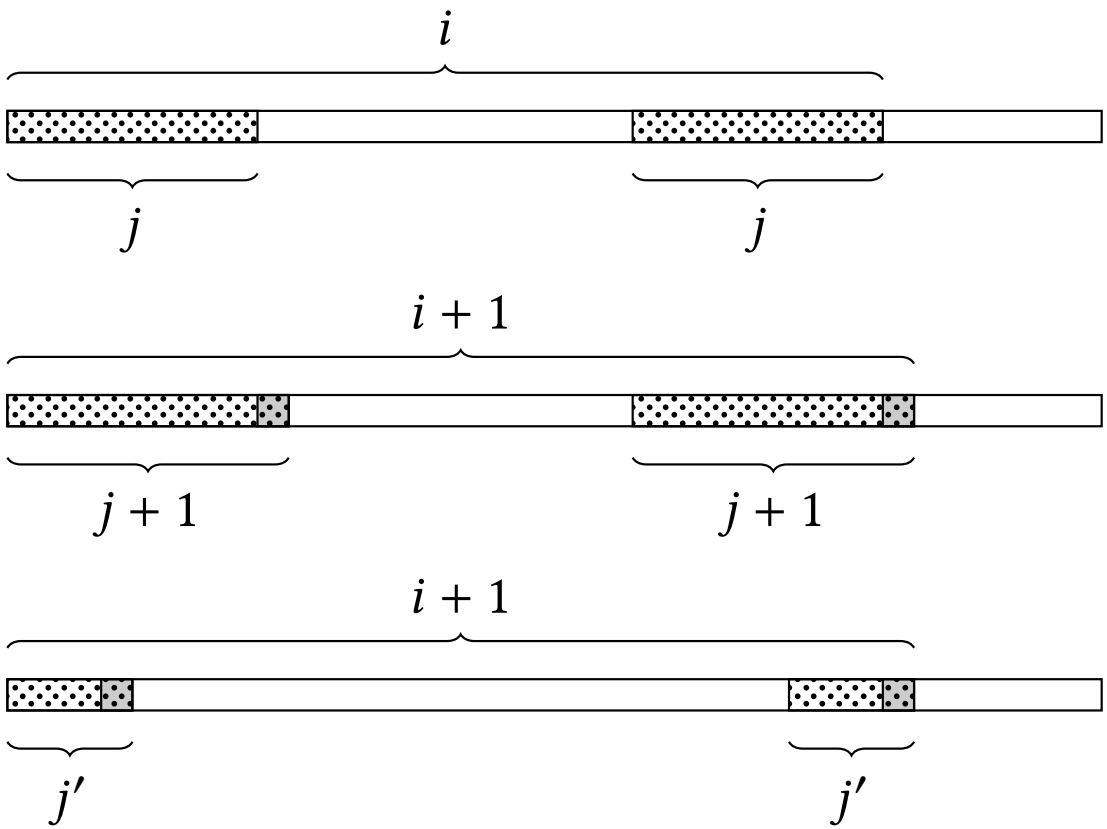
Finding Borders

- If we have already found that a prefix of length i has a border of length j, we check if the prefix of length $i + 1$ has a border of length $j + 1$.
- If this does not happen, we repeat the same process for the immediately shorter border of length $j' < j$, and so on.
- If there is no shorter border, the prefix with length $i + 1$ has border with length zero.

Finding Borders (Contd.)

- We work with increasing prefixes.
- For prefixes of length zero or one, the border length is zero.
- For borders of length i, we check to see if we can extend the existing border by one character. If not, we try shorter borders, while they exist.

Finding Borders Illustrated



Borders Finding Algorithm

Algorithm: Find the borders of a string.

`FindBorders`$(p) \rightarrow b$

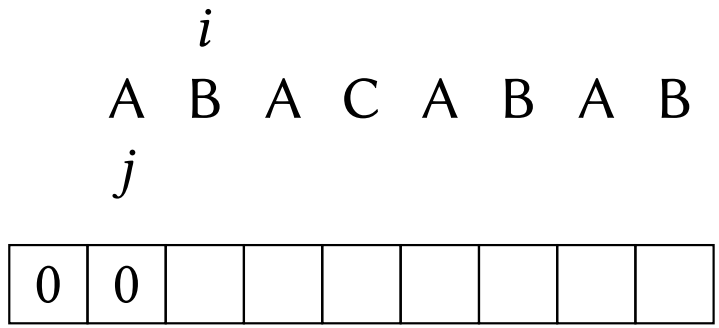
Input: p, a string

Output: b, an array of length $|p| + 1$ containing the lengths of the borders of p; $b[i]$ contains the length of the border of the prefix of p of length i

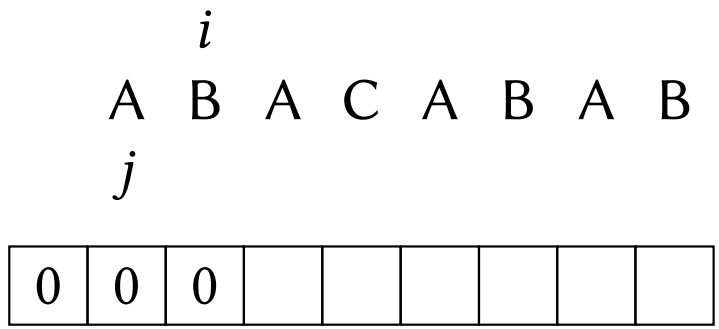
```
1  m ← |p|
2  b ← CreateArray(m + 1)
3  j ← 0
4  b[0] ← j
5  b[1] ← j
6  for i ← 1 to m do
7      while j > 0 and p[j] ≠ p[i] do
8          j ← b[j]
9      if p[j] = p[i] then
10         j ← j + 1
11     b[i + 1] ← j
12 return b
```

Borders Finding Example (1)

$i = 1$
$j = 0$

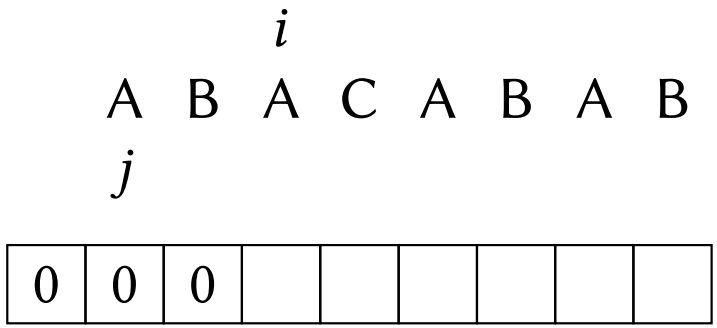


$i = 1$
$j = 0$

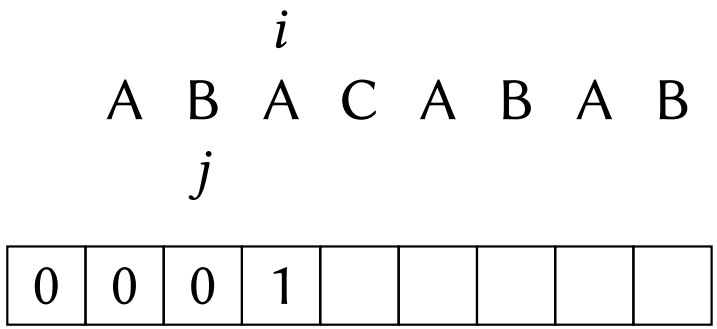


Borders Finding Example (2)

$i = 2$
$j = 0$

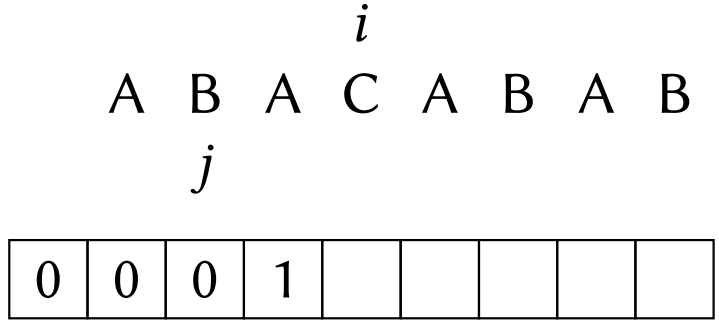


$i = 2$
$j = 1$

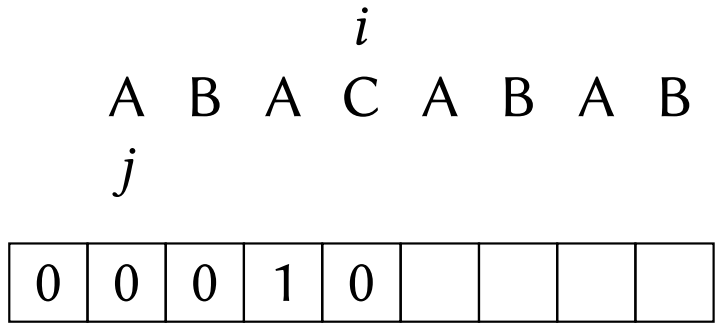


Borders Finding Example (3)

$i = 3$
$j = 1$

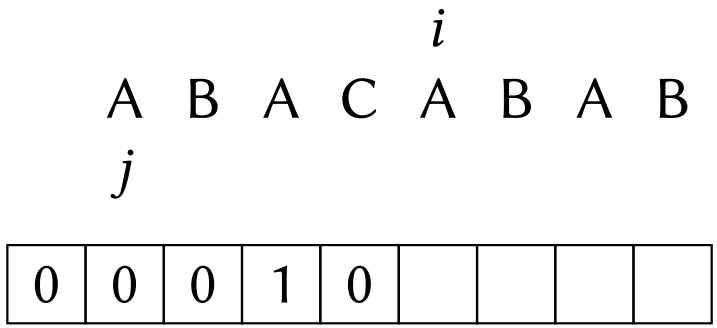


$i = 3$
$j = 0$

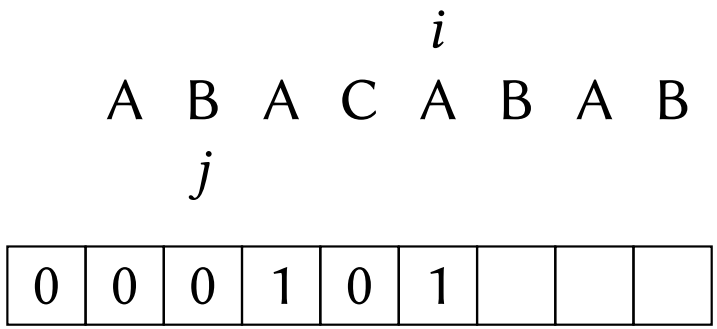


Borders Finding Example (4)

$i = 4$
$j = 0$

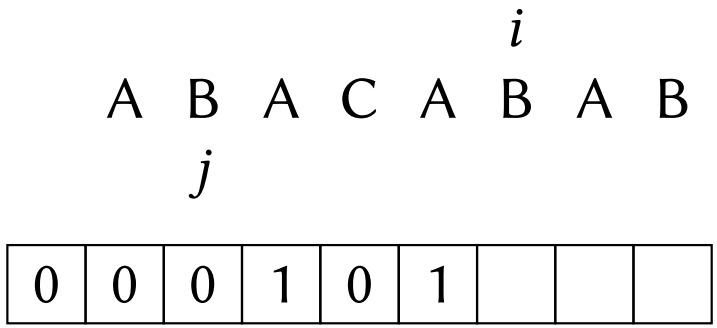


$i = 4$
$j = 1$

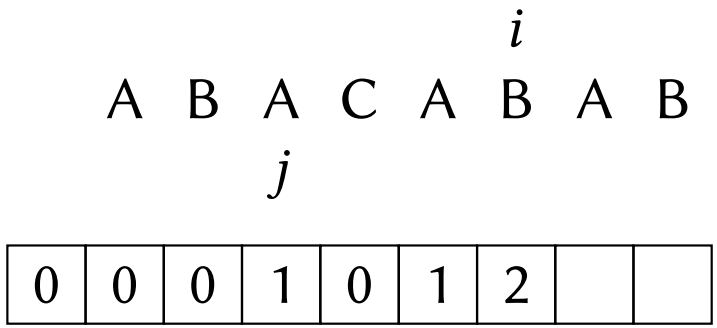


Borders Finding Example (5)

$i = 5$
$j = 1$

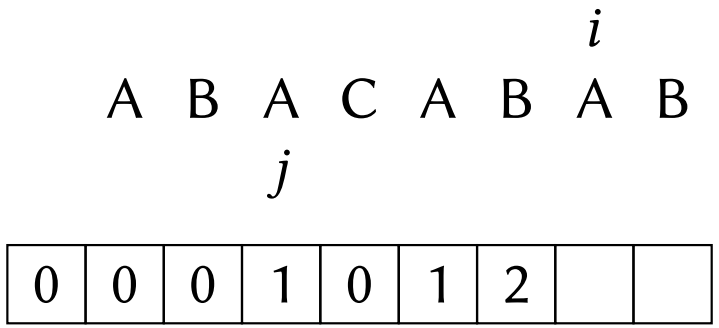


$i = 5$
$j = 2$

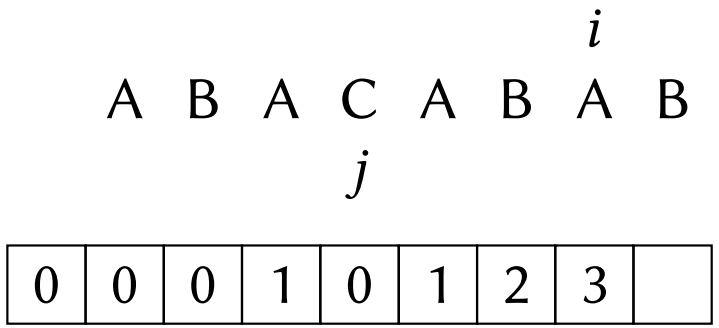


Borders Finding Example (6)

$i = 6$
$j = 2$

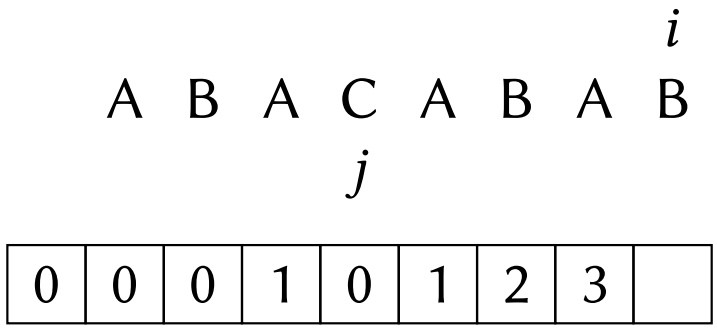


$i = 6$
$j = 3$

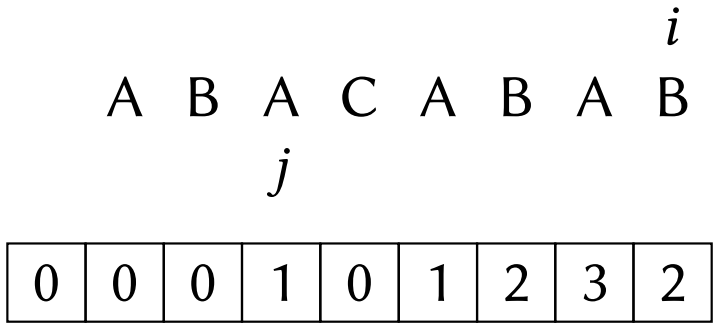


Borders Finding Example (7)

$i = 7$
$j = 3$



$i = 7$
$j = 2$



Knuth-Morris-Pratt Complexity

- The outer loop is executed n times.
- For each one of these iterations we have a number of iterations of the internal loop of lines 7–8.
- In each iteration of the internal loop, j is decreased; the loop is executed while $j > 0$.
- The value j can be increased only ones in each iteration of the outer loop.
- Therefore, the internal loop can be executed up to $O(n)$ times.
- So the overall complexity is $O(2n) = O(n)$, if we do not take into account `FindBorders`.

Knuth-Morris-Pratt Complexity

- Working in the same way, the complexity of `FindBorders` is $O(m)$.
- Therefore, the overall complexity of the algorithm is $O(m + n)$.
- The algorithm has an extra storage cost, as we need an array of size $m + 1$ for the borders, but this is normally not a problem.

Outline

General

- Up to now, we have been examining our text from the left to the right.
- If we change tack and we start examining our text from the right to the left, then we can work with another, simple algorithm, which works well in practice.
- This is the Boyer-Moore-Horspool algorithm, named after its inventors.

Boyer-Moore-Horspool Example (1)

Boyer-Moore-Horspool Example (2)

Boyer-Moore-Horspool Example (3)

Boyer-Moore-Horspool Example

Boyer-Moore-Horspool Example (4)

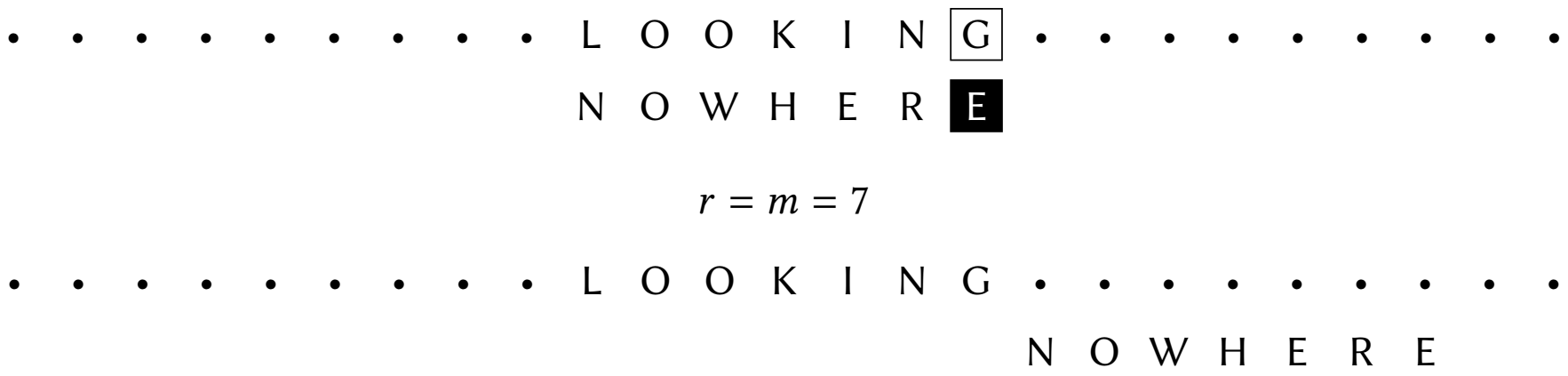
															i				
$i = 12$		A	P	E	S	T	L	E	I	N	T	H	E	K	E	T	T	L	E
														K	E	T	T	L	E

Basic Idea

- We try to match the pattern with the text going from the right to the left.
- If we find a mismatch, the best we can do is slide the pattern to the right until we find a character that matches the current character in the text.
- If there is no such character, we can slide the pattern directly after the mismatched character.
- For the scheme to work we must know how many times to slide each time.

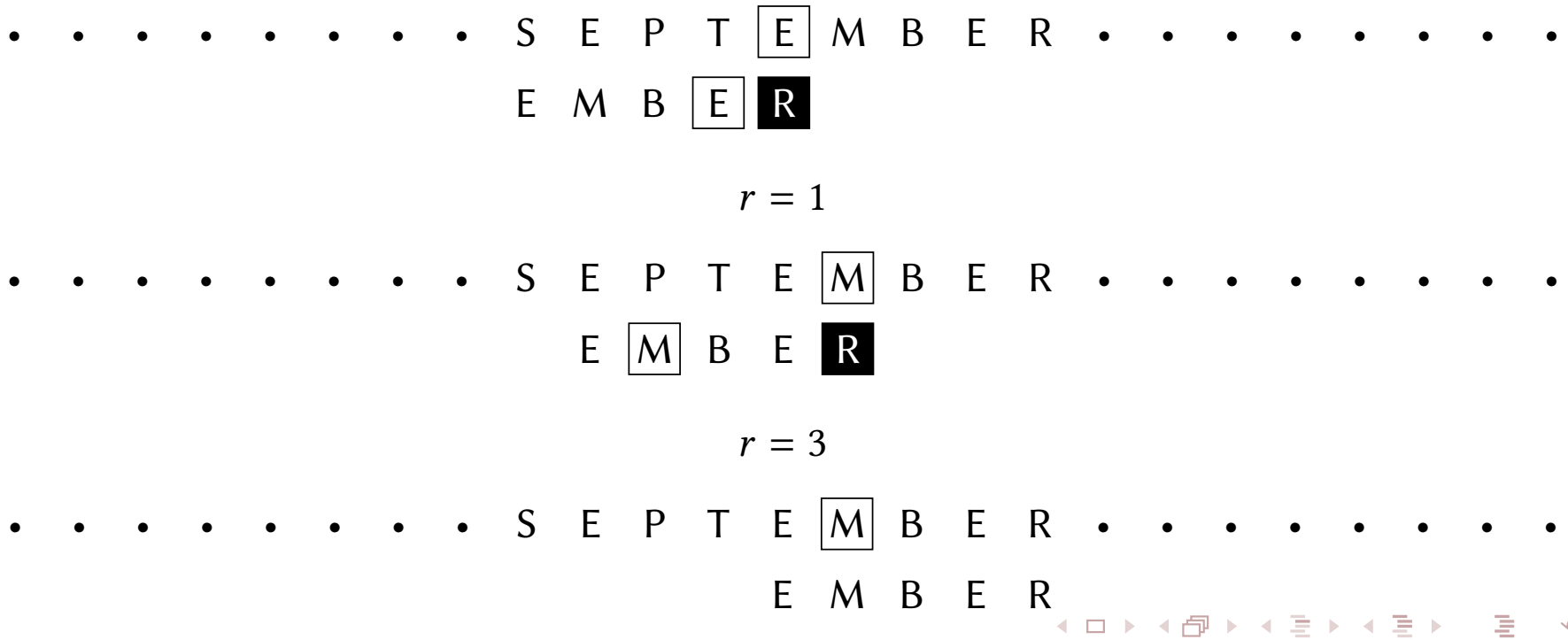
Mismatched Character Not in Pattern

If the mismatched character does not exist in the pattern, we can slide the pattern m positions to the right, where m is the length of the pattern.



Mismatched Character in Pattern

- If there is a mismatch and the mismatched character appears in the pattern at a position $r \geq 1$ counting from the *right*, then we slide the pattern r positions to the right.
- If the character appears more than once, r is the last position, counting from the right.



Rightmost Occurrences Table

- For the algorithm to work, we need to create a table that shows the rightmost occurrences of each character in the pattern.
- The table will be as big as our alphabet.
- For example, if we use the ASCII encoding, it will have 128 positions.
- If the table is called rt, $rt[i]$ is the rightmost position $r \geq 1$ where the i ASCII character appears in the pattern. If it does not appear, $rt[i] = m$, where m is the pattern length.
- If the last character of the pattern appears only once in the pattern, then the corresponding entry is zero. That ensures that a mismatch there will result to a slide over the length of the pattern.

Rightmost Occurrences of Letters in Patterns

Character	K	E	T	T	L	E		E	M	B	E	R
ASCII (decimal)	75	69	84	84	76	69		69	77	66	69	82
ASCII (hexadecimal)	4B	45	54	54	4C	45		45	4D	42	45	52
Occurrence from the right	5	4	2	2	1	4		1	3	2	1	5

Rightmost Occurrences Table for KETTLE

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
1	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
2	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
3	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
4	6	6	6	6	6	4	6	6	6	6	6	5	1	6	6	6
5	6	6	6	6	2	6	6	6	6	6	6	6	6	6	6	6
6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
7	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6

Rightmost Occurrences Table for EMBER

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
1	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
2	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
3	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
4	5	5	2	5	5	1	5	5	5	5	5	5	5	3	5	5
5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
6	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
7	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5

Rightmost Occurrences Creation Algorithm

Algorithm: Create rightmost occurrences table.

`CreateRtOccurrencesTable`$(p, s) \longrightarrow q$

Input: p, a pattern
s, the size of the alphabet

Output: rt an array of size s; for the ith letter of the alphabet, $rt[i]$ will be the index of the rightmost position $r \geq 1$ where the character appears in p, counting from the end of the pattern, or the length of the pattern p otherwise

```
1  rt ← CreateArray(s)
2  m ← |p|
3  for i ← 0 to s do
4      rt[i] ← m
5  for i ← 0 to m − 1 do
6      rt[Ordinal(p[i])] ← m − i − 1
7  return rt
```

Finding Rightmost Occurrences in EMBER

		E	M	B	E	R
i		0	1	2	3	
⋮	5	5	5	5	5	
B	5	5	5	**2**	2	
⋮	5	5	5	5	5	
E	5	**4**	4	4	**1**	
⋮	5	5	5	5	5	
M	5	5	**3**	3	3	
⋮	5	5	5	5	5	
R	5	5	5	5	5	
⋮	5	5	5	5	5	

The Boyer-Moore-Horspool Algorithm

Algorithm: Boyer-Moore-Horspool.

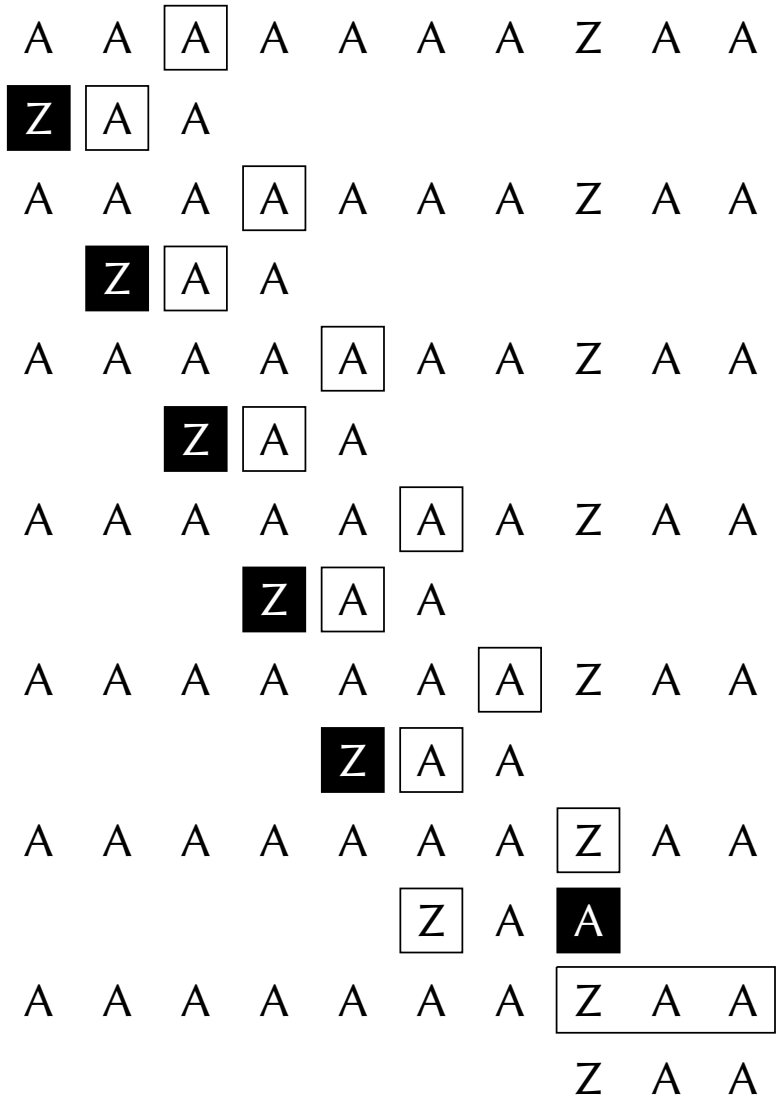
`BoyerMooreHorspool`$(p, t, s) \rightarrow q$

Input: p, a pattern
t, a text
s, the size of the alphabet

Output: q, a queue containing the indices of t where p is found; if p is not found the queue is empty

```
1  q ← CreateQueue()
2  m ← |p|
3  n ← |t|
4  rt ← CreateRtOccurrencesTable(p, s)
5  i ← 0
6  while i ≤ n − m do
7      j ← m − 1
8      while j ≥ 0 and t[i + j] = p[j] do
9          j ← j − 1
10     if j < 0 then
11         Enqueue(q, i)
12     c ← t[i + m − 1]
13     i ← i + rt[Ordinal(c)]
14 return q
```

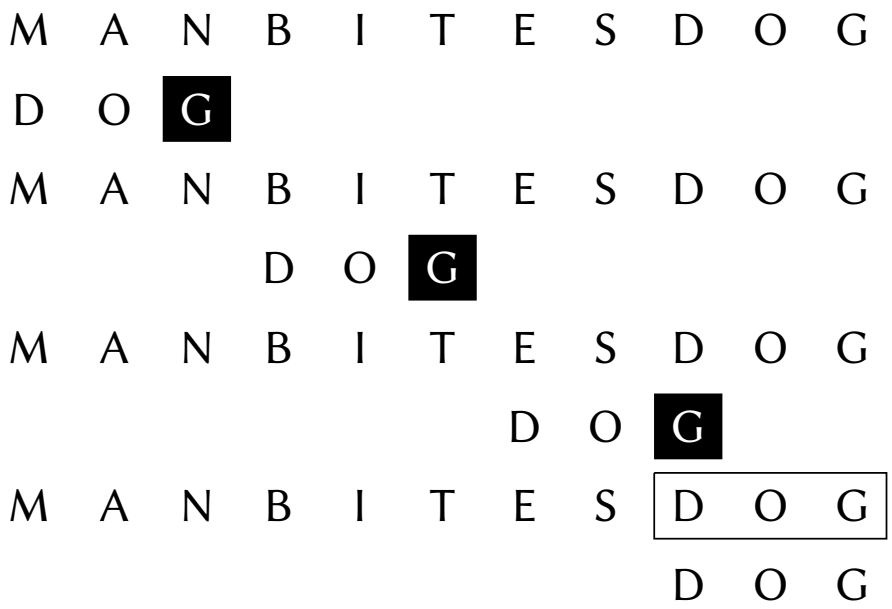
Worst Case


A A A A A A A Z A A
Z A A
A A A A A A A Z A A
Z A A
A A A A A A A Z A A
Z A A
A A A A A A A Z A A
Z A A
A A A A A A A Z A A
Z A A
A A A A A A A Z A A
Z A A
A A A A A A A Z A A
Z A A

Worst Case Analysis

- In the worst case, each time we try to match the pattern we execute $m - 1$ iterations of the inner loop.
- That happens when all characters match from right to left, except for the first character in the pattern.
- So we have $n - m$ iterations of the outer loop *and* $m - 1$ iterations in the inner loop for each of them except the last, in which we have m inner iterations to determine a match.
- Therefore the algorithm has a complexity of $O(nm)$.
- In practice this worst case scenario rarely happens.

Best Case


M A N B I T E S D O G
D O G
M A N B I T E S D O G
D O G
M A N B I T E S D O G
D O G
M A N B I T E S D O G
D O G

Best Case Analysis

- In the best case, the pattern skips repeatedly m characters until it finds a match at the end.
- This gives us a complexity of $O(n/m)$.

Usual Case

- Most cases in practice look like the best case, so the algorithm's complexity in practice is $O(n/m)$.
- The time required to create the rightmost occurrences table is $O(m + s)$, where s is the size of the alphabet.
- That does not change the overall picture, as usually n is much smaller than $m + s$.
- But we do need to store the table.
- For ASCII, the table's size is 128, which is not a problem.
- If, however, our alphabet has thousands of characters, it may be a problem.