CS 663: Pattern Matching Algorithms

Lecturer: Amihood Amir
Scribe: Mohamed Sakeeb Sabakka

Lecture #2
September 1, 2010

Automata Methods for Pattern Matching

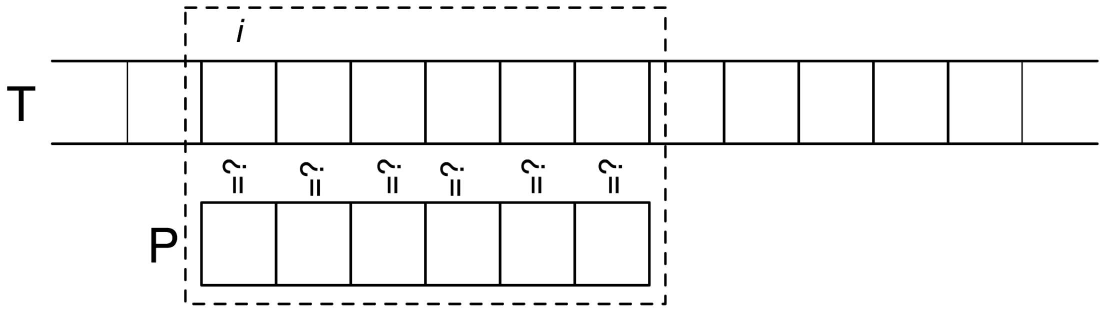
1. Knuth-Morris-Pratt (KMP) algorithm
2. Aho-Corasick algorithm
3. Boyer-Moore Algorithm
4. Bird-Baker Algorithm

Pattern Matching

Pattern matching is about finding all occurrences of a pattern in a given text.

Consider a text T of length T and pattern P of length m.
T = T[0], T[1], T[2], ... T[n]
P = P[0], P[1], P[2], ... P[m].

A naive algorithm to find the match is to iterate through the text T and at each position i, test if the consecutive tokens matches with the pattern P.

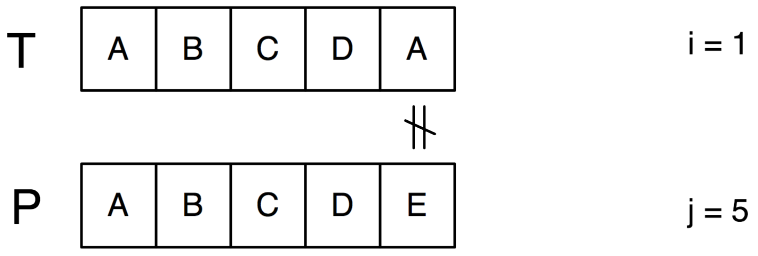


```
for $i \leftarrow 1$ to $n - m + 1$ do
    match ← true
    for $j \leftarrow 1$ to $m$ do
        if $T[i] \neq P[j]$ then
            $match \leftarrow false$
        end
    end
    if $match = true$ then
        output "Match at position", i
    end
end
```

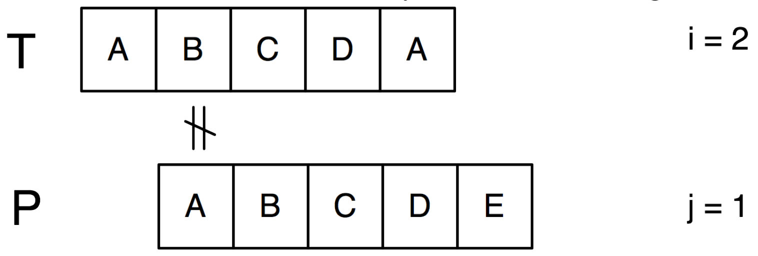
This algorithm has a running time of $O(nm)$.

Now, Lets consider an example.
Let T = ABCDA
and P = ABCDE

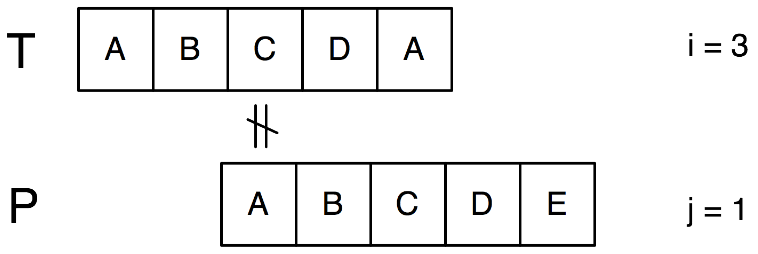
Iterating through the above algorithm, when i = 1, the match = true for j = 1 to 4. But fails for j = 5.

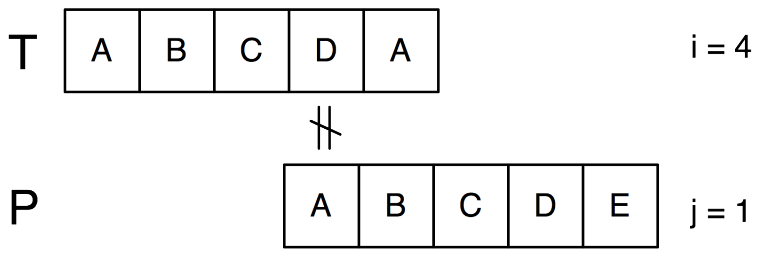


Then we increment i and try to find a match again



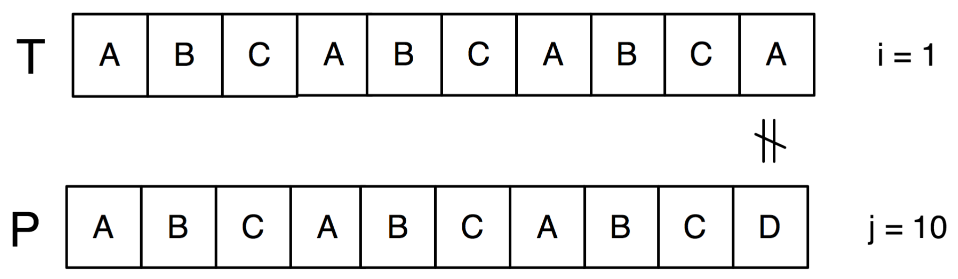
But, does it make sense to try to iterate for all i and try all positions?



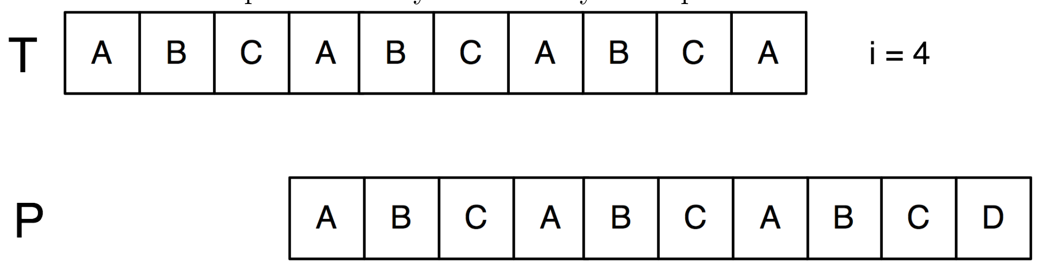


Obviously not. So can we do better?.

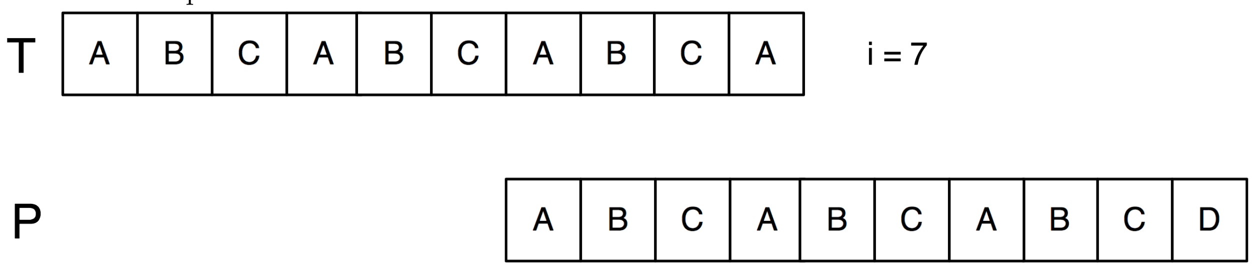
Lets try another example
T = ABCABCABCA
P = ABCABCABCD



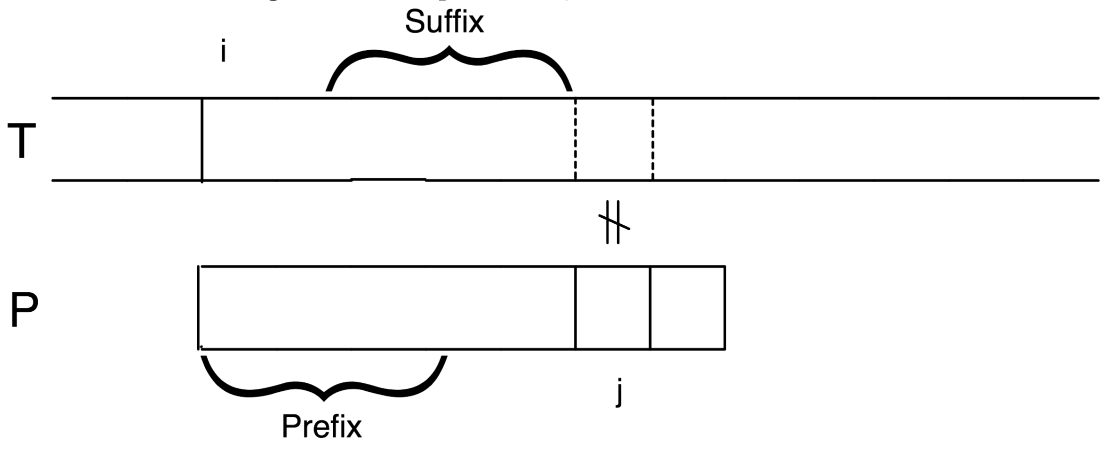
Where is the next position to try?. Intuitively its at position 4.



And the next position is at 7.



So how did we decided on the next position to try? Can we come up with a rule to find the next position to try? Here we are taking the longest proper prefix of the pattern that is a suffix of the text. In the example above ABC is the longest prefix of the pattern which also is the longest suffix at position 1, 4 and 7 of the text.

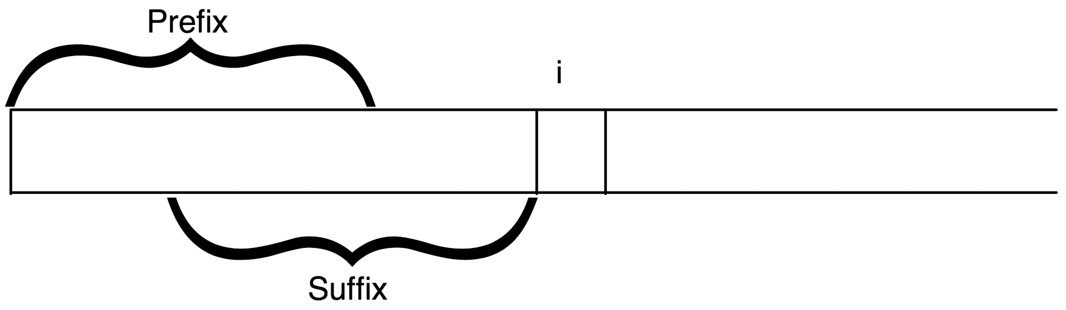


Going with this approach, we need to find the longest prefix of the pattern that matches the longest suffix at each and every location of the text. How fast can we compute this. The solution is to build a table to get this longest proper prefix in constant time. For every pattern location i, get the largest proper pattern prefix that is also a pattern suffix. The size of the table will be $O(m)$

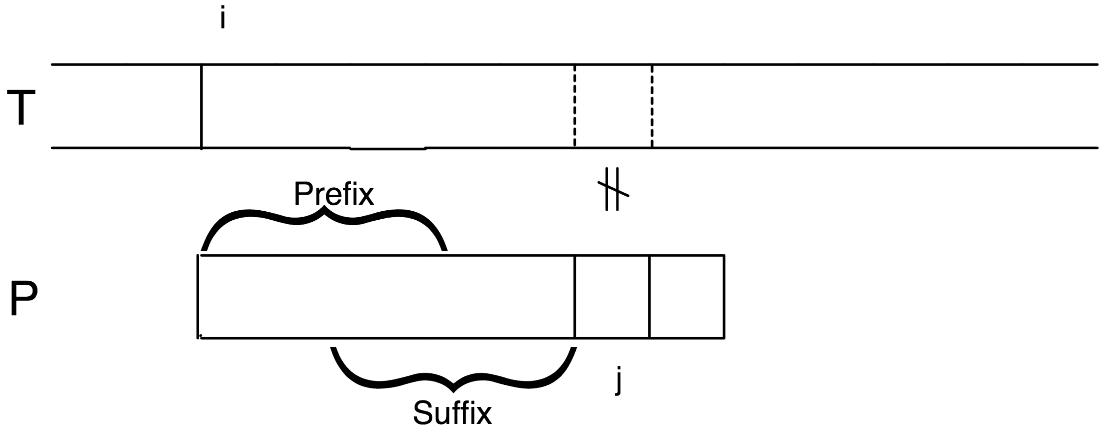
For example, the a pattern ABCBBCAABCB will have the following prefix table.

index	token	prefix
1	A	A
2	B	AB
3	C	ABC
4	B	-
5	B	-
6	C	-
7	A	A
8	A	A
9	B	AB
10	C	ABC
11	B	ABCB

Using the transitivity property of equality, we can see that taking longest proper prefix of the pattern that matches the suffix of the pattern is equivalent as taking the longest suffix of the text.



When the situation is



We have from the table, the largest prefix that is also a pattern suffix

But $P[1] \quad = T[i]$

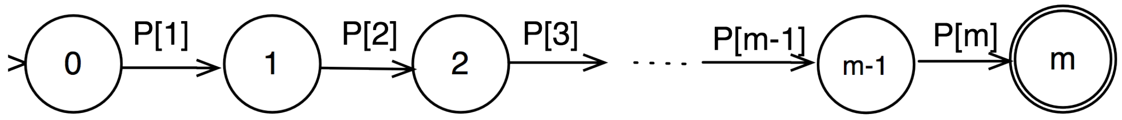
$$
\begin{aligned}
&P[2] &&= T[i+1]\\
&\dots\\
&\dots\\
&P[j-1] &&= T[i+j-2]
\end{aligned}
$$

So, by transitivity, the largest proper prefix of P that is a suffix of P ending at P[j-1] is also the largest prefix of P that is a suffix of T ending at T[i+j-2].

Knuth-Morris-Pratt Automaton Idea(1972)

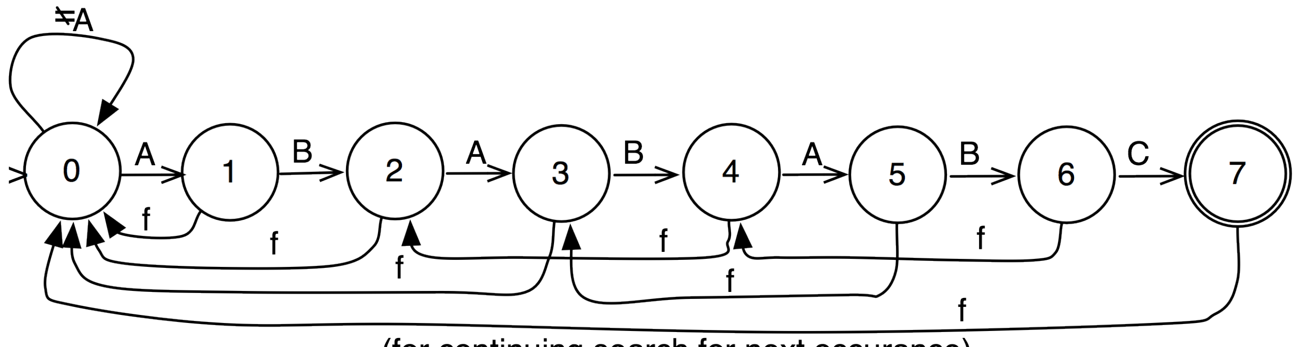
Instead of building a table of prefixes, we can construct and automata whose forward arrows are success links and whose back arrows are failure links.

A success link is added if at a position the next token matches with the proper prefix. The success links for an automation of pattern P = P[1], P[2], ... P[m-1], P[m] is shown below.



The failure links from node i points to node j, where j is the length of the longest proper prefix of P that is a suffix of P ending at P[i].

For example, lets consider the pattern ABABABC.



(for continuing search for next occurance)

For matching the patterns in the text, run the automaton on the text. For success links, move to next text location. For failure links move on link, but stay on the text location. For example, consider:

index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
text	A	B	A	B	C	C	A	A	B	A	B	A	B	B	C

For token C at position 5 of the the text, the automaton will follow the failure link and go back to node 2 while the text will continue from position 5 onwards. For C at position 6 of the text, the automaton follows the failure link again and goes to node 0. For A at position 7 it advances to node 1 but follows failure link for next and goes back to node 0. It continues like this until it reaches the end of the text. If the automaton reaches that last node, ie node 7, we have found a match.

In the automaton, every move on success link was also a move forward in the text. Since we never move back on the text, on success links we move $O(n)$ times forward. On the failure link, we move up to m links backward until the start node on the automaton is reached. On failure links we do not advance on the text. Now it may look like the running time of the algorithm is $O(mn)$. But remember that for every failure link that we follow back, we had to go forward with a success link. So the total number of failure links will never be more than the total number of success links followed. Hence the running time of the algorithm is $O(n)$.

To formalize the proof:
Define a counter F that is initialized to zero.
Increment F by 1 whenever a success link is followed.
Decrement F by 1 when a failure link is followed.

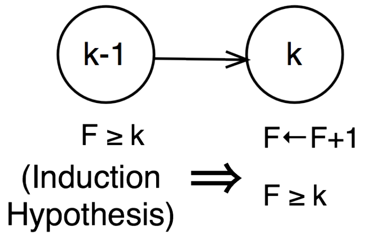
Claim: At any point in the algorithm run, if the automaton is at node k, then $F \geq k$.

Proof: By induction on the number of moves i of the algorithm

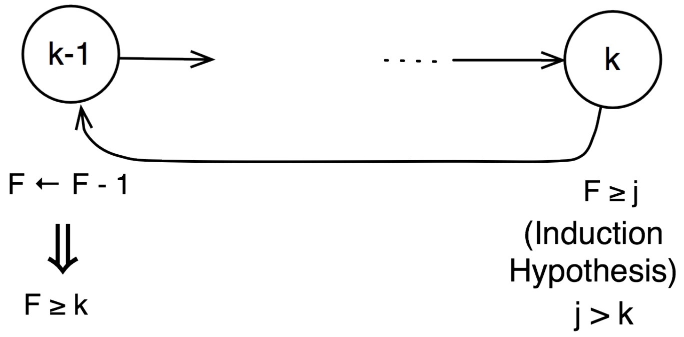
Base case: i = 0:
We are node 0, F = 0
Induction Hypothesis:
After move i, if the automaton is in node k, then $F \geq k$

Prove for move i+1

Case 1: Success link followed in move i+1



Case 2: Failure link followed in move i+1



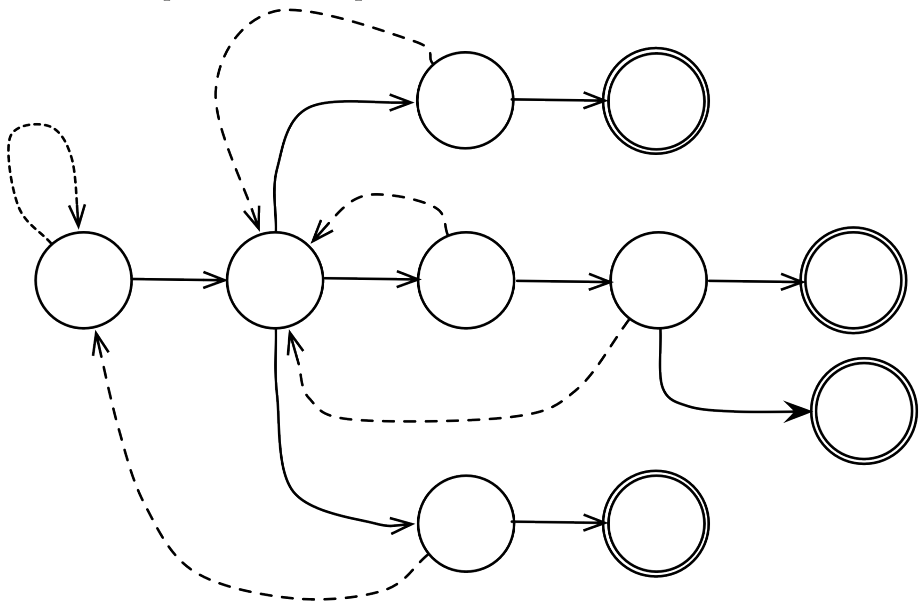
Conclude: F is incremented at most n times and is always non-negetive. There for it is decremented at most n times.

The running time of KMP algorithm is independent of the alphabet used in the pattern or the text.

Assignment 1 The automaton can be created the same way as we used it to find the match. Prove that the automaton can be created in $O(m)$ time. [Due by Sep 15, 2010]

Aho-Corasick algorithm (1975)

This algorithm solves the problem of finding match in a text T, given a set of patterns $\{P_1, P_2, \ldots, P_k\}$. This problem is also known as **dictionary matching**. One very trivial solution is to run KMP algorithm k times. But that will be very inefficient. It will have a run time of $O(nmk)$.
The Aho-Corasick algorithm, further extends the KMP algorithm to use an automaton which can represent all the patterns in the set as shown below



Unlike KMP algorithms, the running time of Aho-Corasick algorithm is dependent on size of the alphabet. If ϵ is the size of the alphabet, at each node of the automaton, the the no of possible links is $log(minimum(\mid \epsilon \mid, k))$. The running time for this automata to run on text of length n will be $O(nlog(minimum(\mid \epsilon \mid, k)))$.

Boyer-Moore Algorithm (1977)

The Boyer-More algorithm successively aligns patter P with Text T and checks if P matches with corresponding tokens in T as in the case of the naive algorithm. Further after the check is complete P is shifted right relative to T just as in the naive algorithm. Further more, it apples some intuite tricks to avoid unnecessary shifts and comparisons. The algorithm is illustrated with example below.

T:	p	a	t	t	e	r	n	-	m	a	t	c	h	i	n	g	a	l	g	o	r	i	t	h	m
P:	a	l	g	o	r	i	t	h	m																

For any alignment of P with T the Boyer-Moore algorithm checks for an occurrence of P by scanning characters from right to left rather than from left to right as in the naive algorithm. The first character being compared is $'m'$ which aligns with the pattern. So it moves to next character which is a $'-'$ character. Since $'-'$ is not a token in the pattern, its comparison with any character in P will only result in failure and hence we can safely shift the pattern to the right of the mismatched character (in this case $'-'$).

T:	p	a	t	t	e	r	n	-	m	a	t	c	h	i	n	g	a	l	g	o	r	i	t	h	m
P:									a	l	g	o	r	i	t	h	m								

The next token being compared is $'m'$ against $'a'$. Here $'m'$ is present in the pattern and its shift index computed (distance from right) is 8. So we need to shift at least by 8 positions to find a match.

T:	p	a	t	t	e	r	n	-	m	a	t	c	h	i	n	g	a	l	g	o	r	i	t	h	m
P:																	a	l	g	o	r	i	t	h	m

At this position the pattern matches with the text.

The worst case running time of the algorithm is $O(n)$.
The best case running time is $O\left(\frac{n}{m}\right)$

Bird-Baker Algorithm

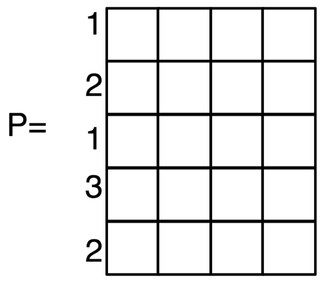
Bird-Baker Algorithm use for two-dimensional pattern matching where given the input text as $T[1 \ldots n][1 \ldots n]$ and pattern as $P[1 \ldots m][1 \ldots m]$, we need to find all occurrences of P in T. This kind of pattern matching is useful for computer vision and related fields.

A naive algorithm to solve this problem is to run KMP on each row of the text for each row of the pattern and to check for matches of successive rows on pattern at the same index. This will be really inefficient and will result in a running time of $O(n^2m^2)$.

KMP for row 1 = $O(n^2)$
KMP for row 2 = $O(n^2)$
KMP for row 3 = $O(n^2)$
...
KMP for row m = $O(n^2)$

which gives $O(n^2m)$. Now we need to run KMP once again for each row of the pattern to find if it has all the successive rows at same index. So the total running time will be $O((n^2m)^2)$.

The Bird-Baker algorithm enhances the naive algorithm by assigning a number p_k to each **distinct** row in the pattern. Then it builds an $n \times n$ array R such that if $T[i, j-m+1 \ldots j]$ matches with with the row marked with p_k in the pattern, $R[i, j] = p_k$.



The real improvement in the Bird-Baker algorithm is that it treat each row in the pattern P as separate patterns to make an automaton as in Aho-Corasick method.

R =

2				2			
1							
		2					
		3		1			
2				2			
3				1			
1				3			
2				2			

To build the Aho-Corasick automaton it takes $O(|\ \epsilon\ |\ m^2)$ time.
To run Aho-Corasick on each row, it takes $O(n^2)$.
To run KMP on each column to find if the pattern sequence appear exactly, it takes $O(n^2)$.

So the total running time of the Bird-Baker algorithm will be $O(n^2 + |\ \sigma\ |\ m^2)$.

Note: The problem of getting rid of ϵ component was an open problem till 1991. It was addressed by Duelling algorithm. It achieves this by treating the 2-dimensional pattern as a one dimensional array.