CS4311
Design and Analysis of Algorithms

Tutorial: KMP Algorithm

About this tutorial

- Introduce String Matching problem
- Knuth-Morris-Pratt (KMP) algorithm

String Matching

- Let T[0..n-1] be a text of length n
- Let P[0..p-1] be a pattern of length p
- Can we find all locations in T that P occurs?

- E.g., T = `bacbababababacbb`

  P = `ababa`

Here, P occurs at positions 4 and 6 in T

Brute Force Approach

- The easiest way to find the locations where P occurs in T is as follows:

```
For each position of T
    Check if P occurs at that position
```

- Running time: worst-case $O(np)$

Brute Force Approach

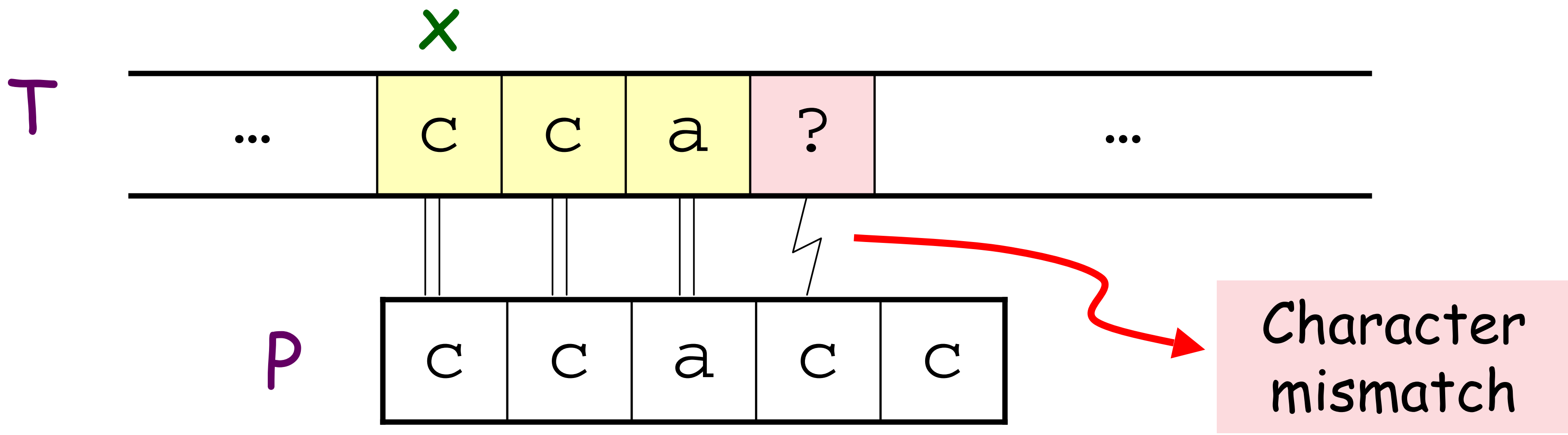
- In the previous algorithm, after we check if P occurs at position x, we start over for the match of P at position x+1

- But we may learn some information during the checking of position x

  ➔ may help to speed up later checking

Brute Force Approach

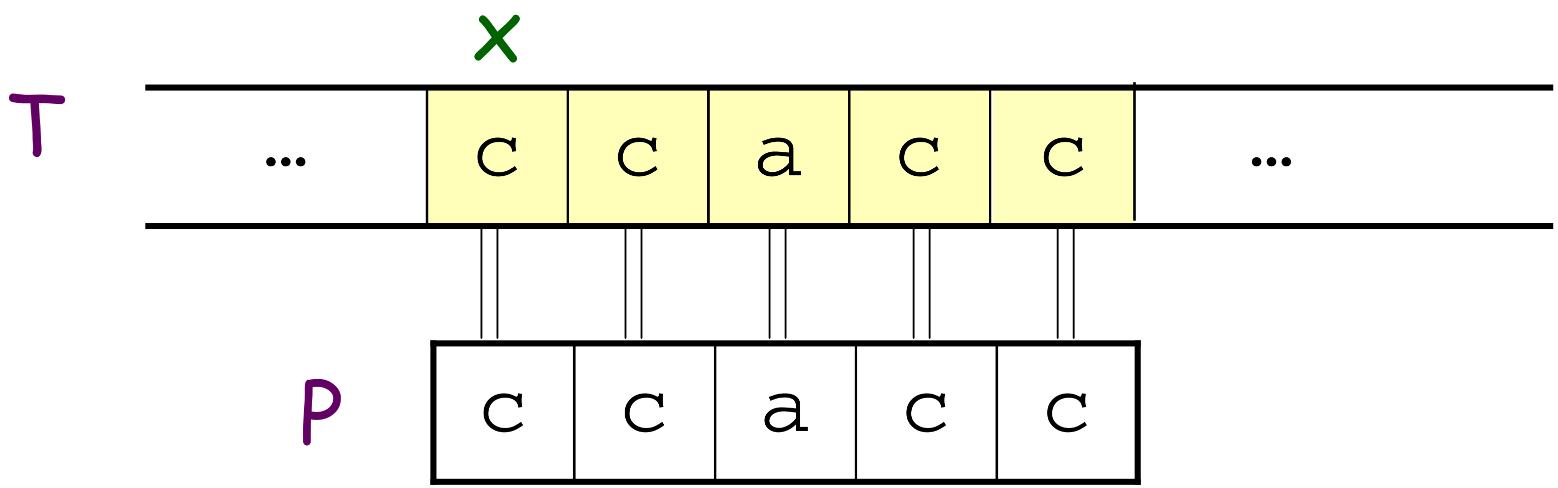
E.g., suppose when we check if P occurs at position x, we get the following scenario:



Can P occur in positions x + 1 or x + 2 ?

Brute Force Approach

How about this case?



Can P occur in positions x+1, x+2, or x+3?

Key Observation

Lemma:

Suppose P has matched k chars with T[x…]

That is, P[0..k-1] == T[x..x+k-1],

Then, for any 0 < r < k,

if T[x+r…x+k-1] is not a prefix of P,

P cannot occur at position x + r

How Many Positions to Skip ?

- When T[x..] gets a first mismatch after matching k chars with P, so that we know

  P[0..k-1] == T[x..x+k-1]

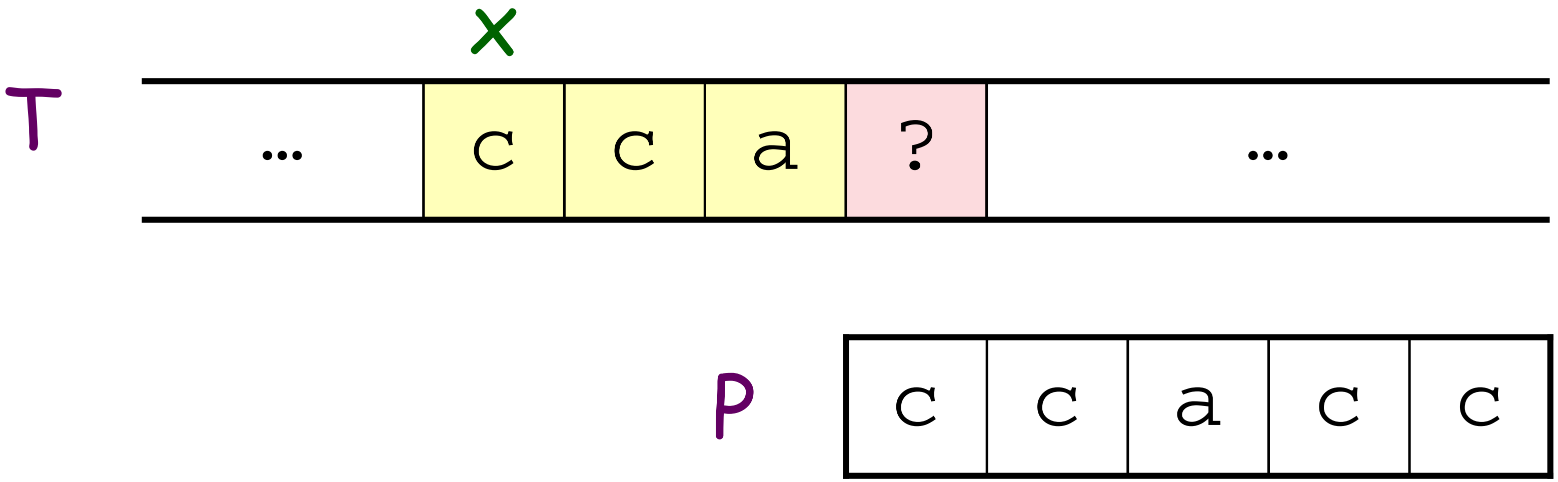
  we can restart the next checking at the leftmost position x+r such that

  T[x+r..x+k-1] is a prefix of P

- Thus "skipping" r positions

Key Observation

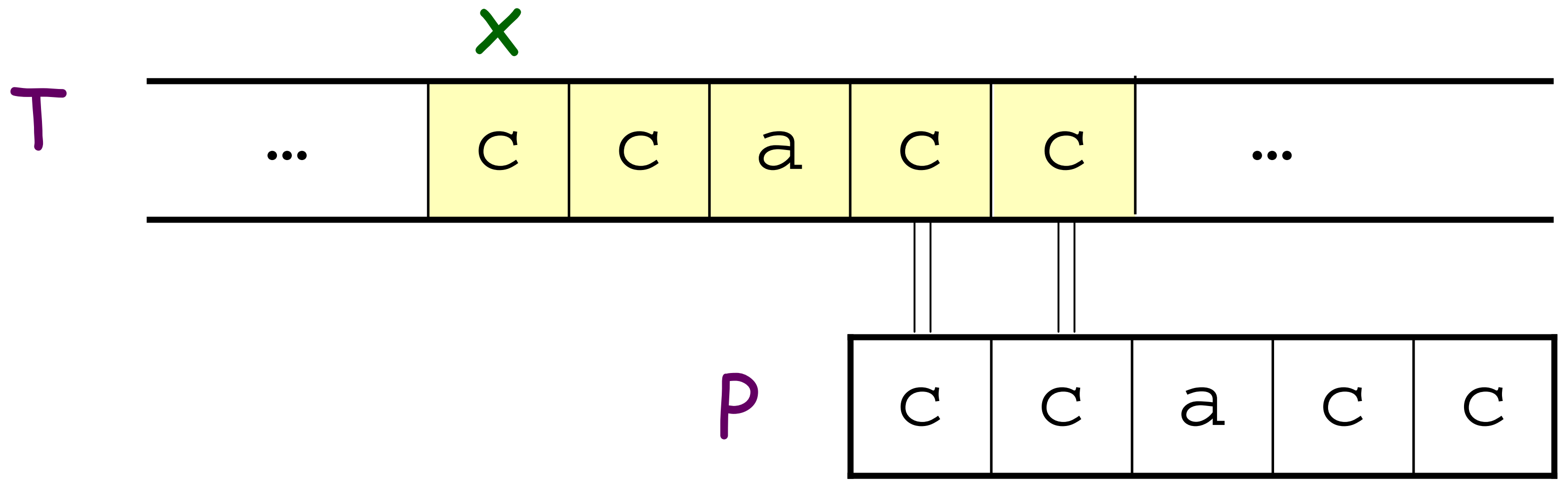
E.g., in our first example,



next checking can restart at pos x+3

Key Observation

In our second example,



next checking can restart at pos x+3

Finding Desired r

- We observe that

  T[x+r..x+k-1] == P[r..k-1]

- So to find the desired r, we need the smallest r such that (why smallest?)

  P[r..k-1] is a prefix of P

- What does that mean ??

Finding Desired r (Example 1)

P

c	c	a	c	c

When k = 3, we ask:

prefix of P ?
No ...

	c	a		

prefix of P ?
No ...

		a		

Thus, we set r=3

Finding Desired r (Example 2)

P

c	c	a	c	c

When k = 5 (what does that mean??), we ask:

prefix of P ?
No ...

	c	a	c	c

prefix of P ?
No ...

		a	c	c

prefix of P ?
Yes ! (r=3)

			c	c

Finding Desired r

- For each k,

smallest r with P[r..k-1] == prefix of P

implies

P[r..k-1] is longest such prefix

- We now define a function π, called **prefix function**, such that

$\pi(k)$ = length of such P[r..k-1]

KMP Algorithm

- The KMP algorithm relies on the prefix function to locate all occurrences of P in $O(n)$ time → optimal !
- Next, we assume that the prefix function is already computed
  - We first describe a simplified version and then the actual KMP
- Finally, we show how to get prefix function

Simplified Version

```
Set x = 0;
while (x < n-p+1) {
   1. Match T with P at position x ;
   2. Let k = #matched chars ;
   3. if ( k == p )  output "match at x" ;
   4. Update x = x + k - π(k) ;
}
```

Skipping positions

What is the worst-case running time ?

How can we improve ?

- In simplified version, inside the **while** loop,

  Line 1 restarts matching (every char of) T with P from position x

- In fact, we know that after "skipping",

  the first $\pi(k)$ chars are already matched

- What if we take advantage of this ??

KMP Algorithm

Set $x = 0$; $k = 0$;

while ($x < n-p+1$) {

1. Match T with P at position x
   but starting from $k+1^{th}$ position;
2. Update k = #matched chars;
3. if ($k == p$) output "match at x" ;
4. Update $x = x + k - \pi(k)$;
5. Update $k = \pi(k)$;

}

k keeps track of #matched chars

Running Time

- The running time comes from four parts:
  1. Mis/matching a char of T with P (Line 1)
  2. Updating the position x (Line 4)
  3. Output match (Line 3)
  4. Updating k (Line 2, Line 5)

Since each char is matched once, and x increases for each mismatch

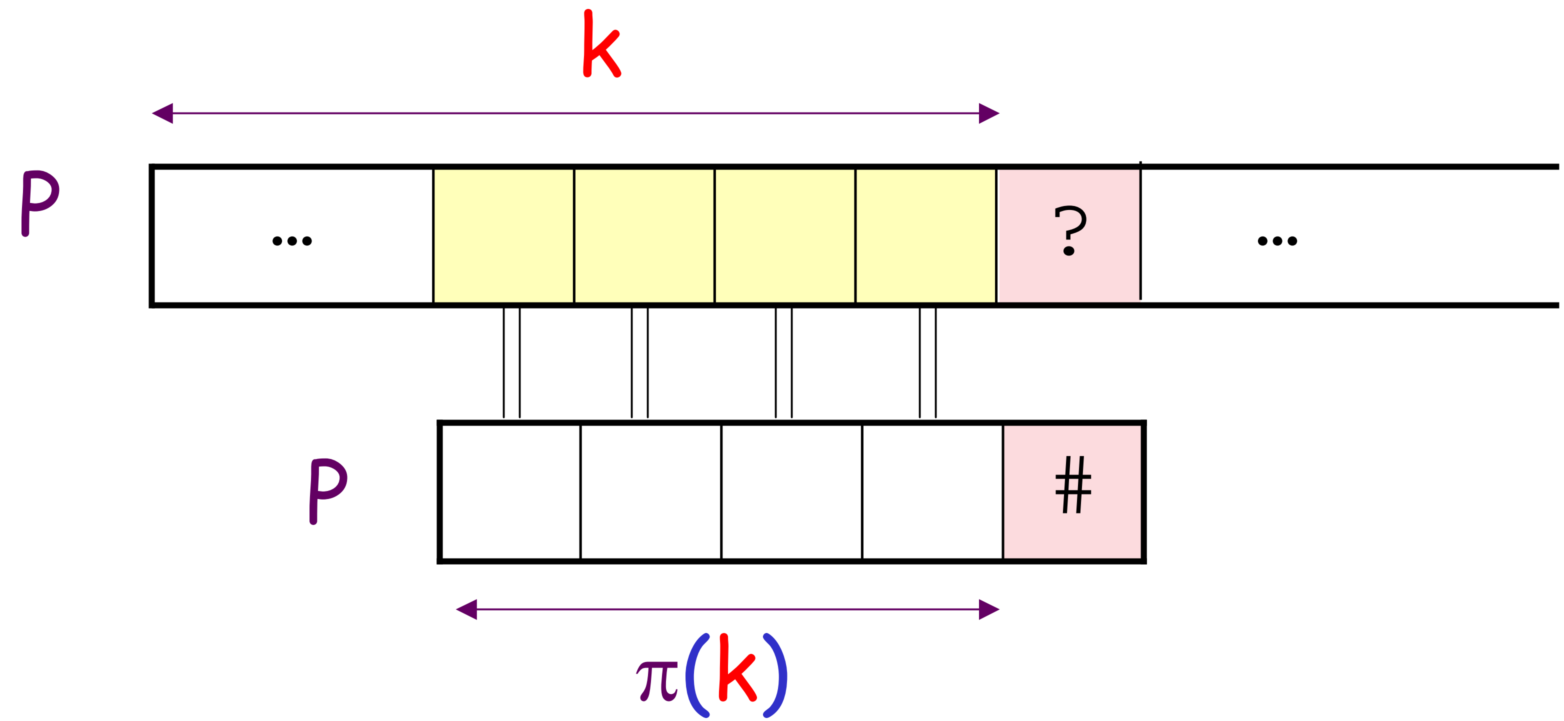
→ in total O(n) time

Computing Prefix Function

- It remains to compute the prefix function
- In fact, it can be computed incrementally
  (finding $\pi(1)$, then $\pi(2)$, then $\pi(3)$, and so on)
- For instance, suppose we have obtained $\pi(1), \pi(2), \ldots, \pi(k)$ already
  ➔ How can we compute $\pi(k+1)$?

Computing $\pi(k+1)$

We know that a prefix of length $\pi(k)$, P[0.. $\pi(k)$-1], is the longest prefix matching the suffix of P[0..k-1]

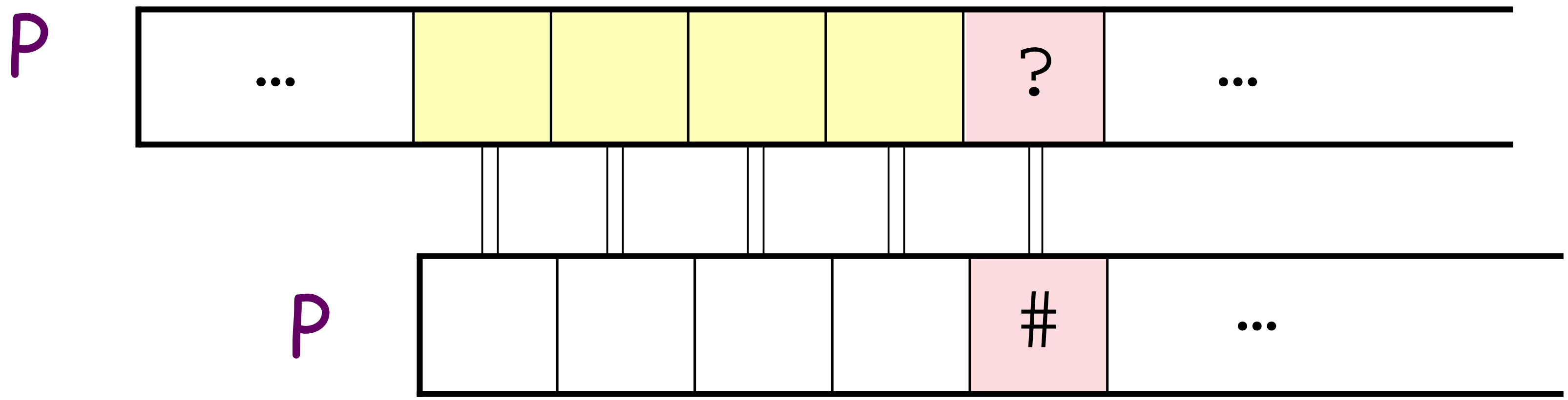


Computing $\pi(k+1)$

What if the next corresponding chars,

$P[\pi(k)]$ and $P[k]$

are the same ??

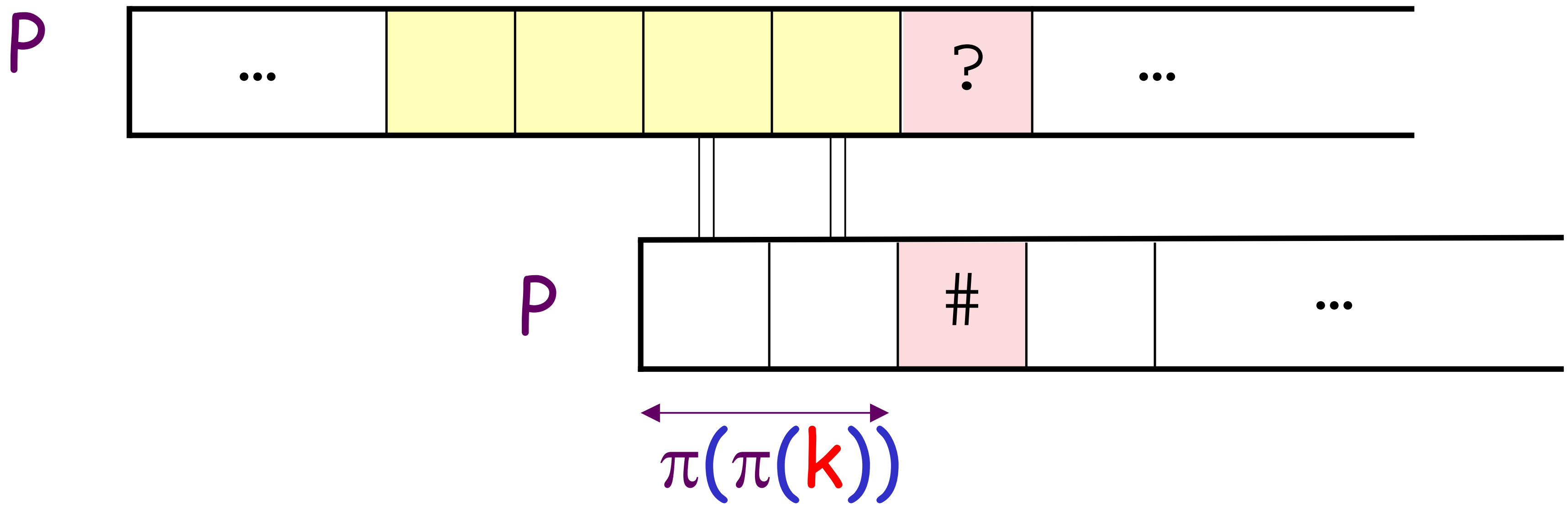


If same, $\pi(k+1) = \pi(k) + 1$ (prove by contradiction)

Computing π(k+1)

Else P[π(k)] and P[k] are different

Then, we should move P below rightwards to search for the next longest prefix of P matching the suffix of P[0..k-1]

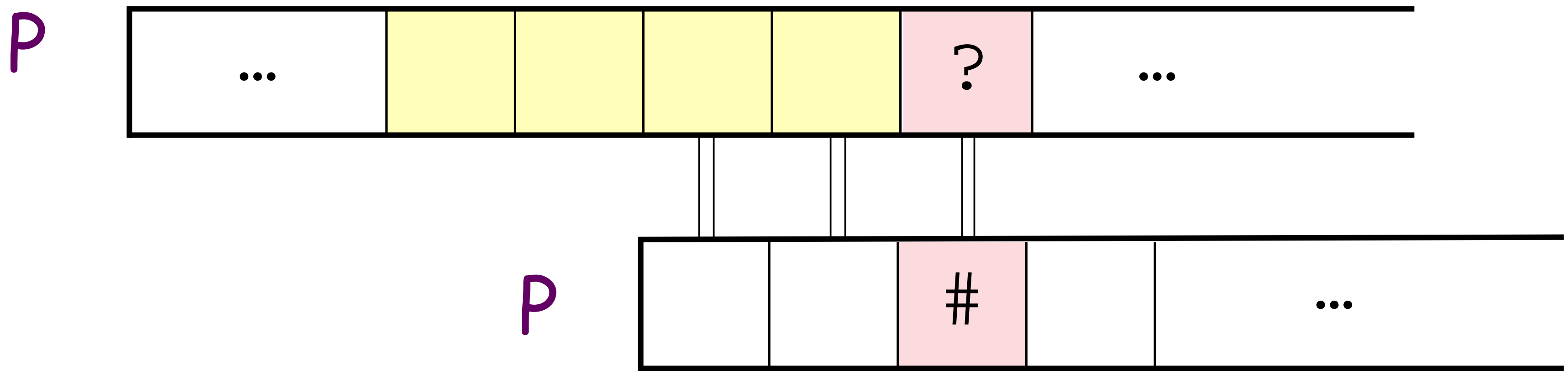


Computing $\pi(k+1)$

What if the next corresponding chars,

$P[\pi(\pi(k))]$ and $P[k]$

are the same ??



If same, $\pi(k+1) = \pi(\pi(k)) + 1$ (prove by contradiction)

Computing π(k+1)

- Else P[π(π(k))] and P[k] are different, and we see that we can repeat the procedure and obtain π(k+1) as soon as we find:

the longest prefix of P matching the suffix of P[0..k-1], with its next char == P[k]

- same procedure as string matching algo
- Total time to compute π : O(p) time

since (1) at most P matches, and

(2) P below moves rightwards for each mismatch