

---

# Searching for a pattern. Knuth-Morris-Pratt

---

Lecture 2.

---

# Motivation

*“In a very real sense, molecular biology is all about sequences. It tries to reduce complex biochemical phenomena to interaction between defined sequences”*

G. Von Heijne. Sequence analysis in molecular biology: treasure trove or trivial pursuit. Academic press, 1987

---

---

# Examples

- Finding the overlaps during the sequence assembly
  - Finding *STS* – Sequence Tagged Sites – unique sequences used to map the positions of the fragments in the genome
  - Finding *EST* – Expressed Sequence Tags – STSs of protein-coding DNA – to locate genes inside the entire sequenced genome
-

---

# Useful definitions

- A *string*  $S$  of length  $N$  is an ordered list of  $N$  elements written contiguously from left to right
  - The elements are called *symbols* or *characters*
  - $S[i..j]$  is a contiguous *substring* of  $S$  starting at position  $i$  and ending at position  $j$  of  $S$
  - $S[1..j]$  is a *prefix* of  $S$  starting at position 1 and ending at position  $j$
  - $S[i..N]$  is a *suffix* of  $S$  starting at position  $i$  and running till the last character of  $S$
  - $S[i..j]$  is an *empty string* if  $i > j$
  - A *proper* substring, prefix, suffix of  $S$  is respectively a substring, prefix, suffix that is neither the entire string  $S$  nor the empty string
-

---

# Pattern matching problem

- Given a string  $P$  (of length  $M$ ) called the *pattern* and a longer string  $T$  (of length  $N$ ) called the *text*, find all occurrences, if any, of pattern  $P$  in text  $T$
-

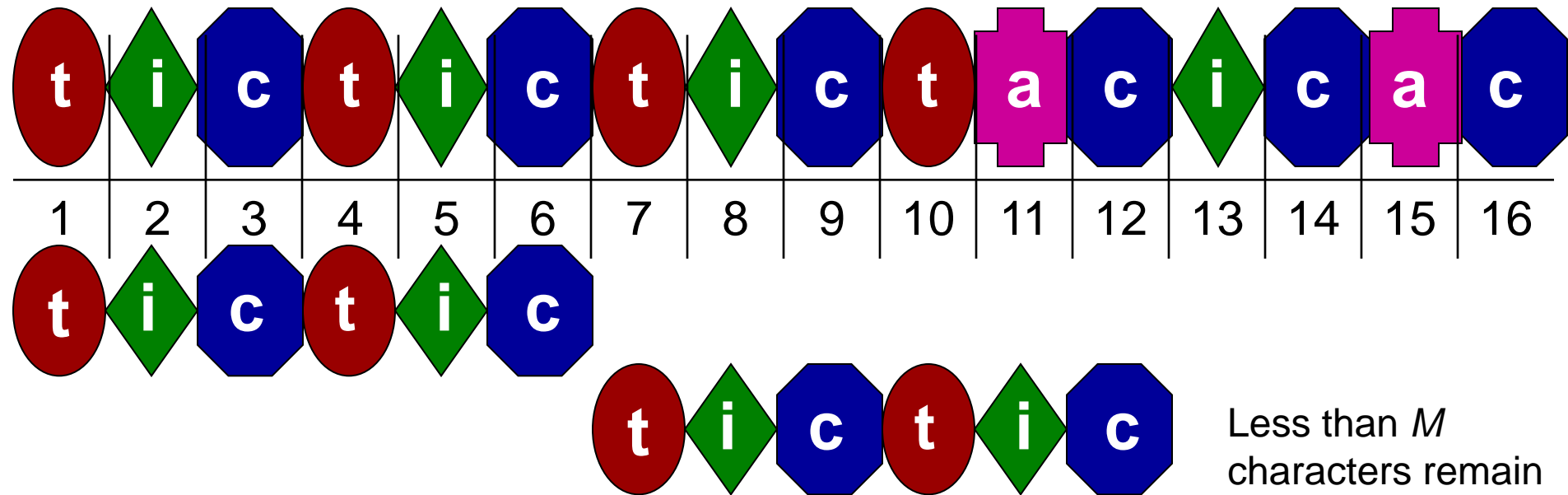
# Naïve method – time complexity

- Naïve method is to compare the characters of the pattern starting from each of  $N$  positions of the text
- In the worst case, it requires  $O(MN)$  character comparisons, exactly  $M(N-M+1)$ , for example, for  $T=aaaaaaaaaa$  ( $N=10$ ) and  $P=aaa$  ( $M=3$ ) there are 24 character comparisons

# Naïve method – time complexity

- In the worst case, we start from each position  $i$  of  $T$  (there are  $N$  such positions), and for each such position check, in the worst case, all  $M$  characters of  $P$
- A standard fetching time from sequential RAM is 358 MB values per second ([ref](#)).
- If we have 10 random sets of sequenced fragments from the 3 GB-length human genome, then we need to search the text of a total size  $3 \cdot 10^{10}$ , which can be sequentially accessed with approximately  $3 \cdot 10^8$  values per second. We will spend 100 seconds on a linear time algorithm, but for the worst case we need to multiply it by the value of  $M$ , which can be as large as 800.
- *Grep* search program (based on a linear-time algorithm), for example, requires about 2 minutes when searching for a string of length 10 in a 3 GB text (on an average desktop machine).
- We want the pattern search algorithm to perform in a *linear time*

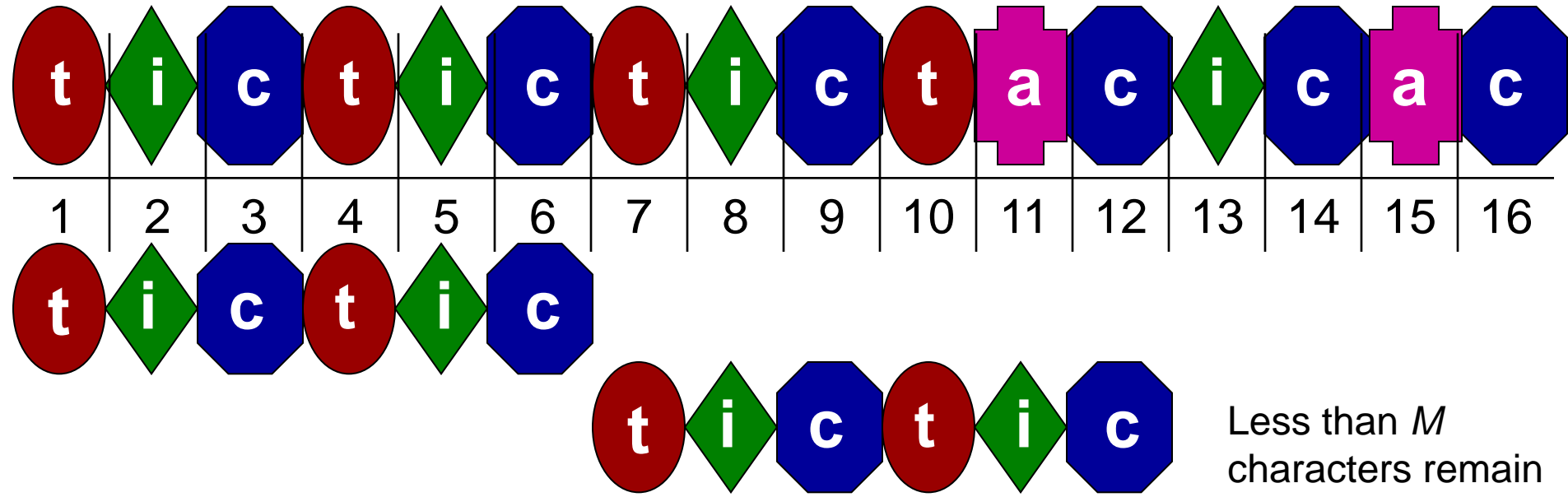
Our dream goal: each character of  $T$  is accessed only once



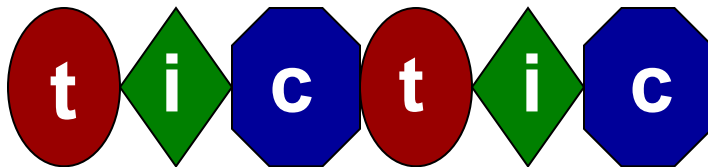
Is this algorithm correct?



# Incorrect algorithm



No, we have missed an occurrence of P starting at position 4

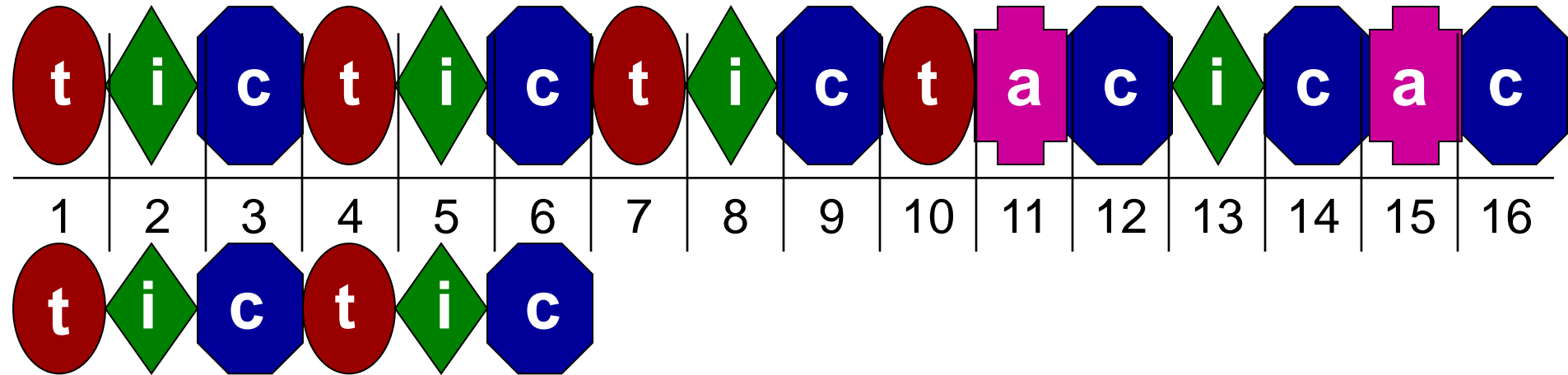


---

# Knuth-Morris-Pratt (KMP) idea

- When we have aligned the prefix of  $P$  with  $k$  characters of  $T$ , we know what characters are in  $T$  up to the current position (they are equal to those of the prefix  $P[1\dots k]$  of  $P$ )
  - From this information we can deduce the place where to start the next comparison
-

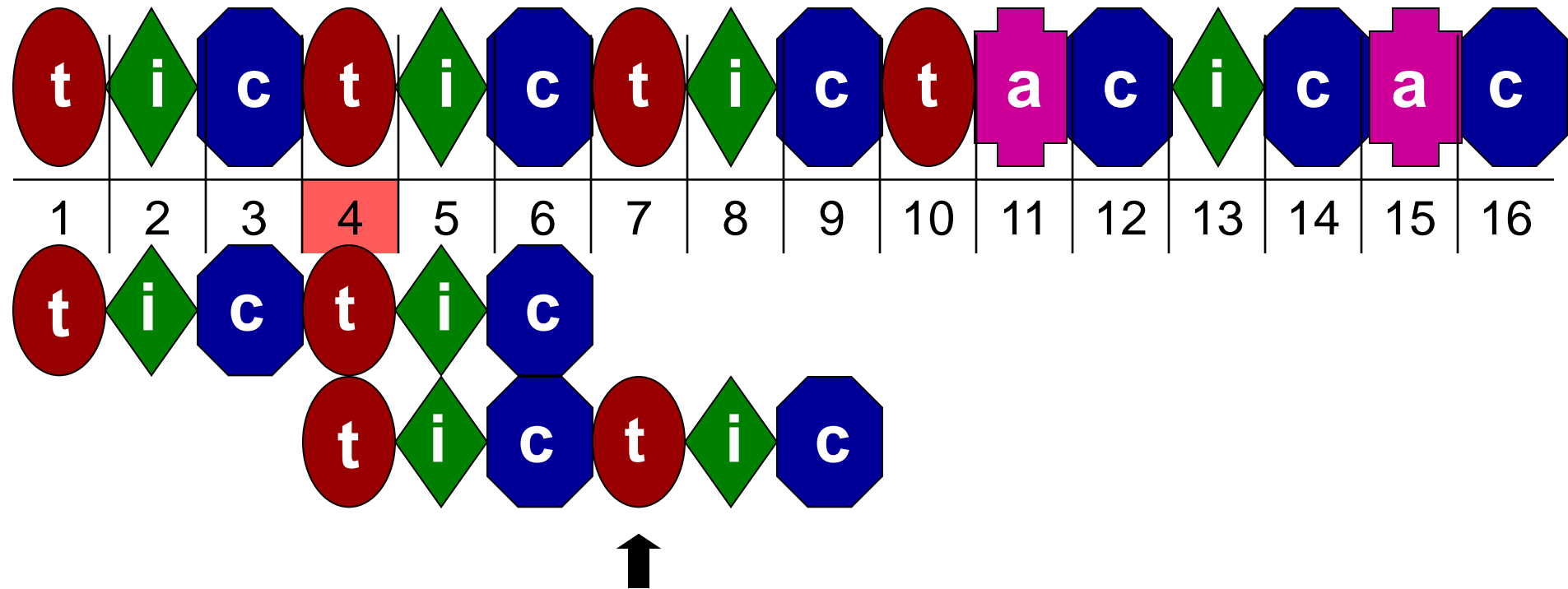
# KMP intuition



We have aligned 6 characters

The next occurrence of a pattern has to start with *tic* and we know that the last characters of a match were *tic*, since the suffix of  $P$  starting at position 4 is equal to a prefix of  $P$  of length 3

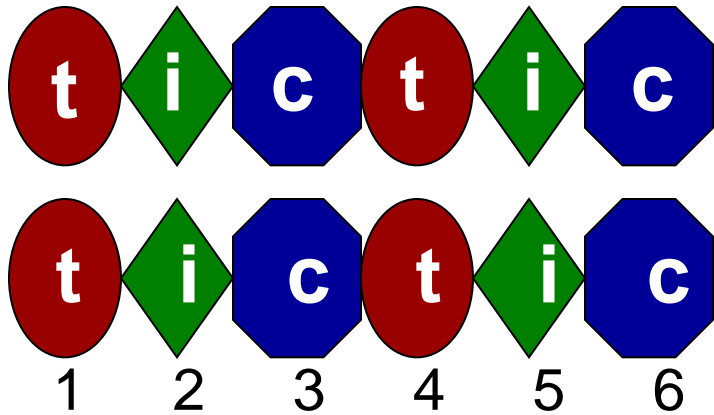
# KMP intuition



Therefore we can set a start of the next comparison to 3 positions backwards from the current position (red cell), and we don't need to compare the first 3 characters of  $P$  again, since we know that they match

Thus, we can continue the comparison from the next character of  $P$  (and  $T$ )

# KMP intuition – overlap function for $P$

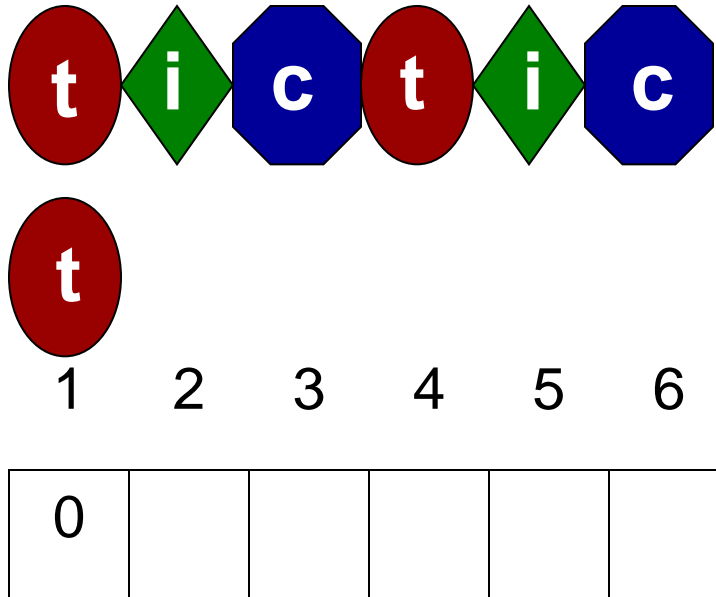


In order to know where to position the start of the next comparison, we need to know the values of an *overlap function* for  $P$ , namely:

For each position  $j$  in  $P$ , the maximal length of a substring which is at the same time a proper prefix of  $P$  and the proper suffix of substring  $P[1, j]$ .

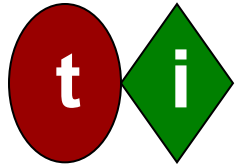
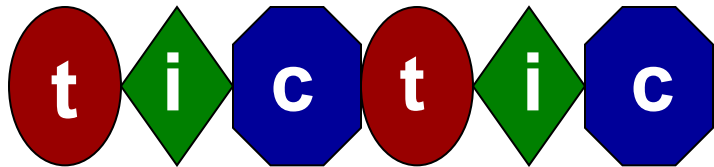
Before we start the search, we need to compute an overlap function for  $P$  – we need to preprocess pattern  $P$

# KMP intuition – overlap function for P



For  $j=1$ ,  $OF=0$  ( $t$  is not a proper suffix of a substring  $t$ , but the entire  $t$ )

# KMP intuition – overlap function for P

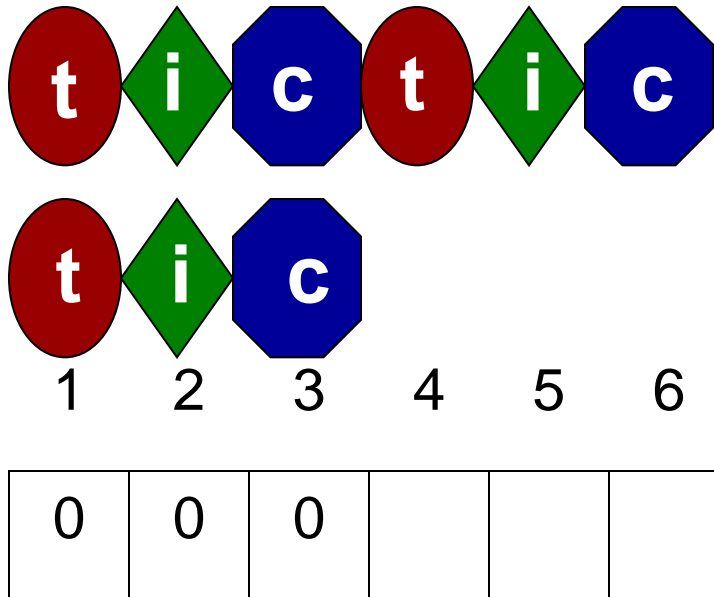


1 2 3 4 5 6

0	0				
---	---	--	--	--	--

For  $j=2$ ,  $OF=0$  (the only proper suffix of  $ti$ , the suffix  $i$ , does not have any overlap with the prefix  $t$  of  $ti$ )

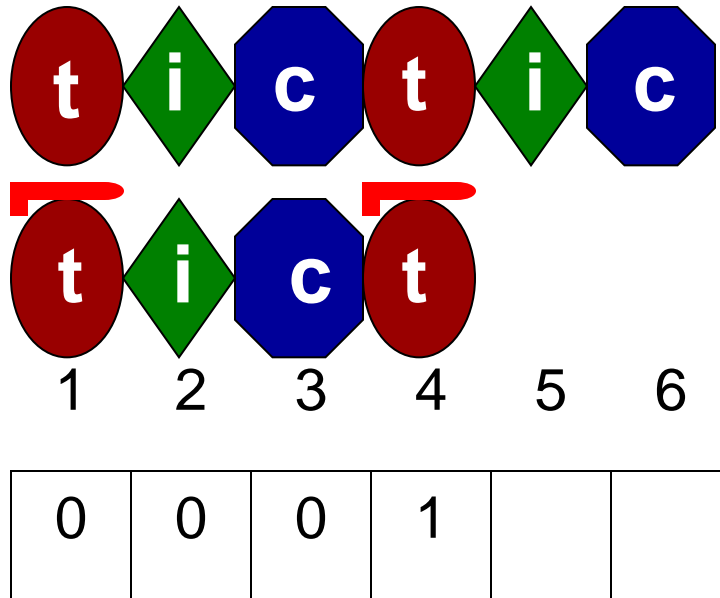
# KMP intuition – overlap function for P



For  $j=3$ ,  $OF=0$  (suffixes *ic*, *c* do not have an overlap)

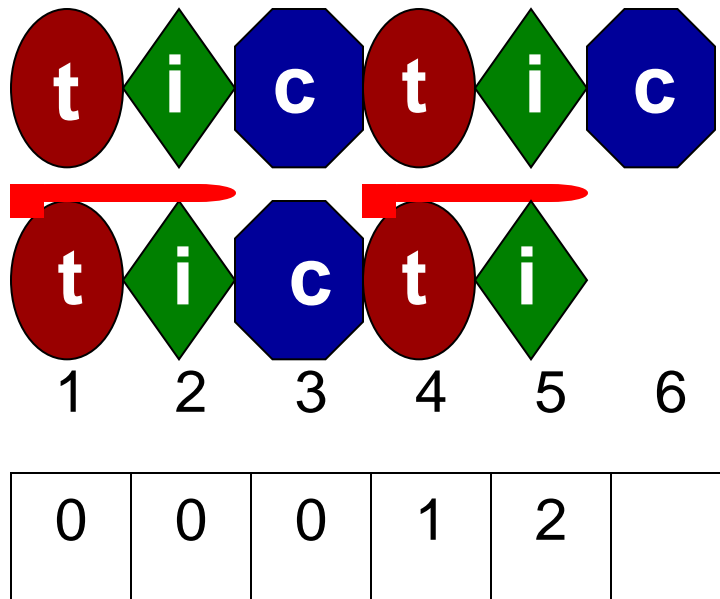


# KMP intuition – overlap function for P



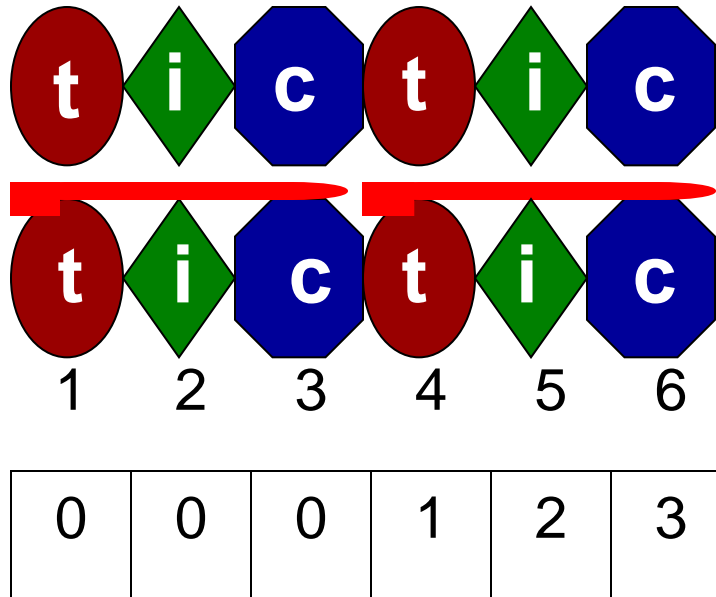
For  $j=4$ ,  $OF=1$  ( $t$  is a proper suffix of a substring  $tict$ , and the prefix of P)

# KMP intuition – overlap function for P



For  $j=5$ ,  $OF=2$  (*ti* is a proper suffix of a substring *ticti*, and the prefix of P)

# KMP intuition – overlap function for P

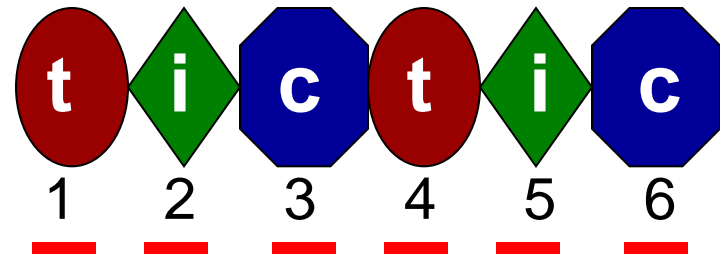
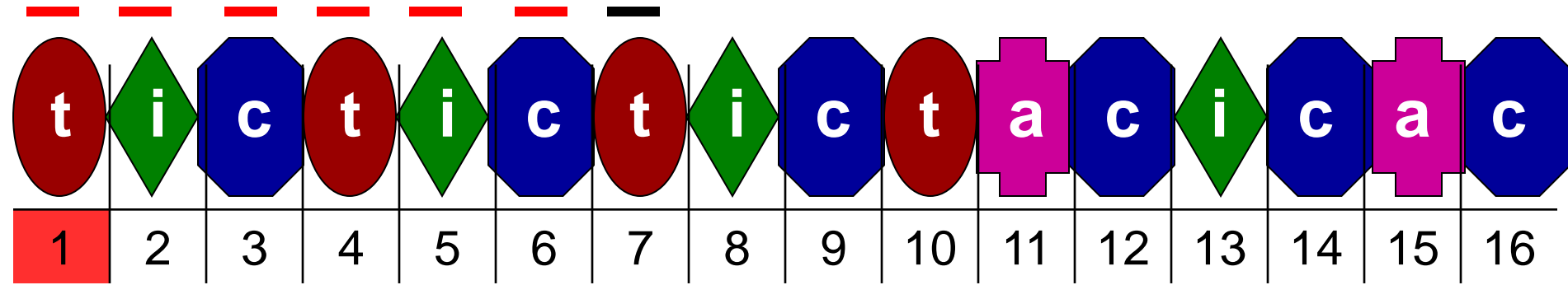


For  $j=6$ ,  $OF=3$  ( $tic$  is a proper suffix of a substring  $tictic$ , and the prefix of  $P$ )

Assume, for now, that the  $OF$  values for  $P$  are computed

# KMP search

$i=7$



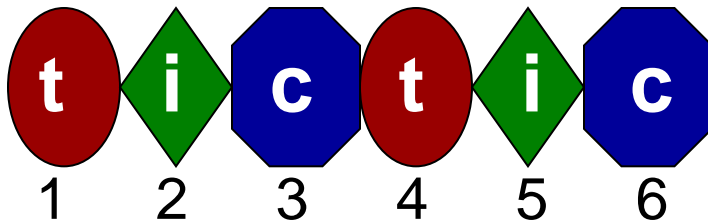
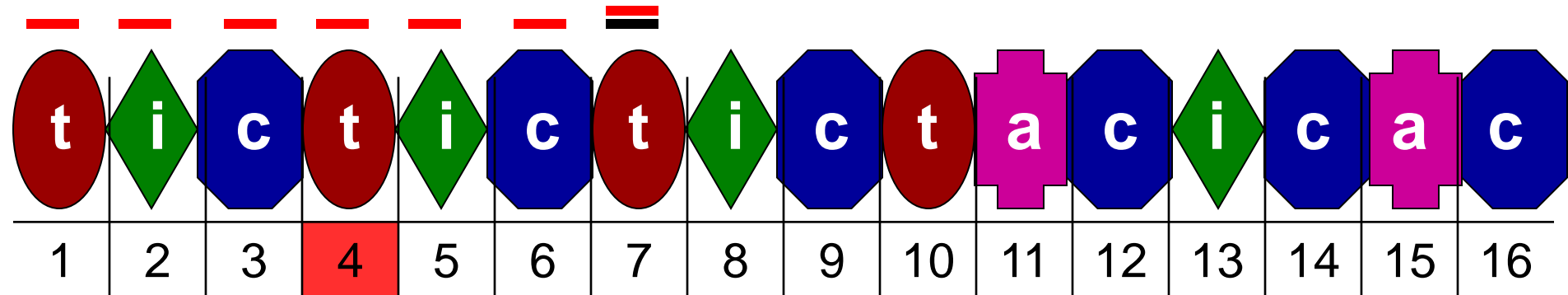
$j=7$

Report **1**

0	0	0	1	2	3
---	---	---	---	---	---

Consult  $OF(6)=3$  it tells how many positions backward from  $i$  the next comparison starts:  $k=i-OF(j-1)$

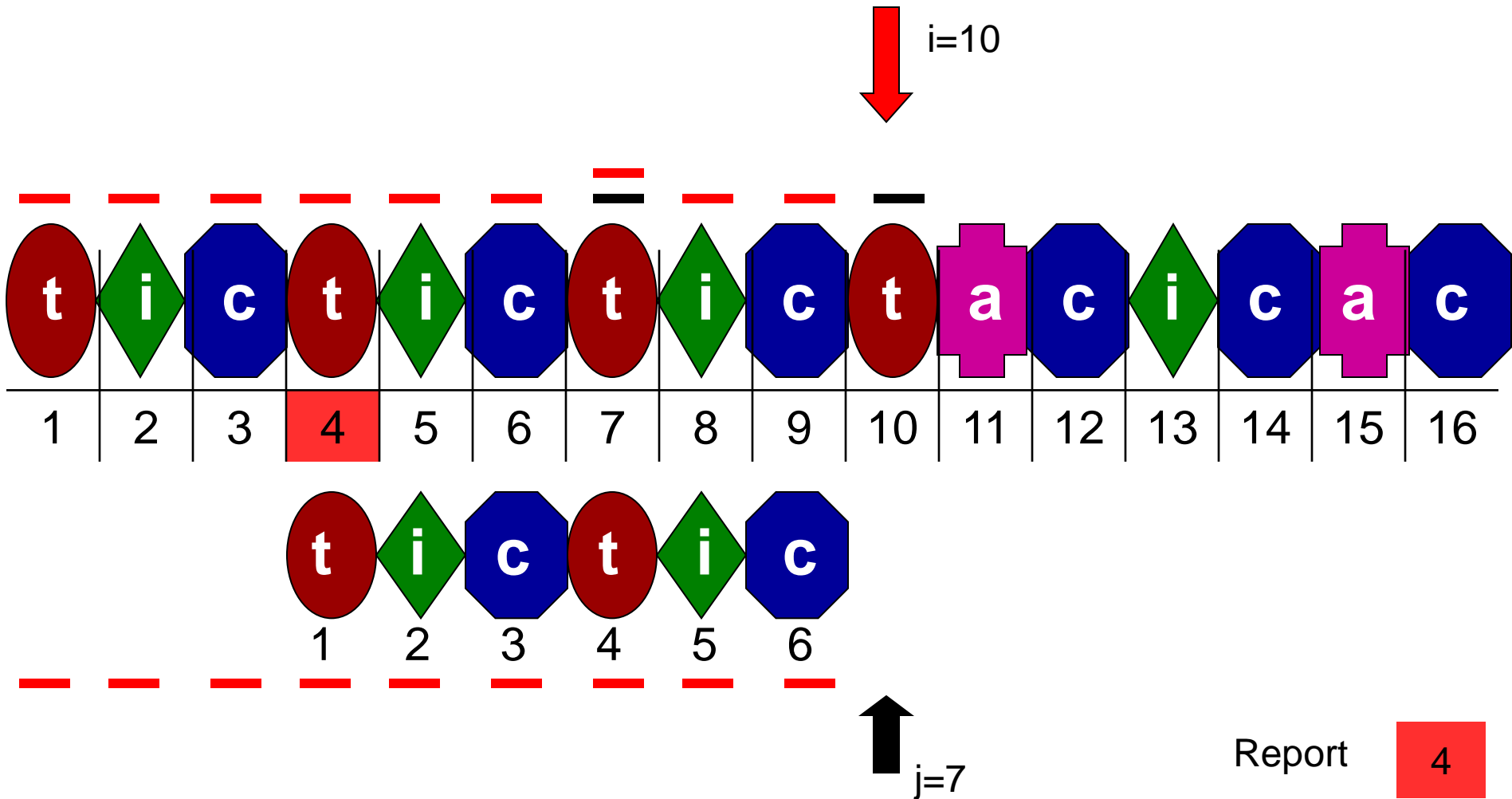
# KMP search



0	0	0	1	2	3
---	---	---	---	---	---

No need to compare these 3 characters, we know that they match – we just compared them

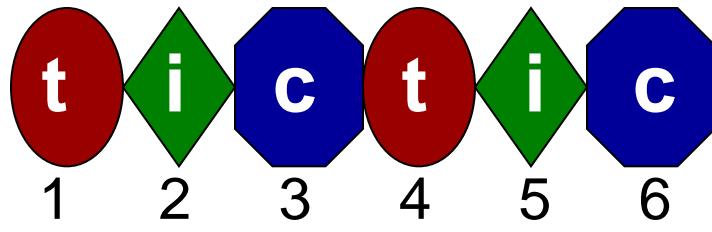
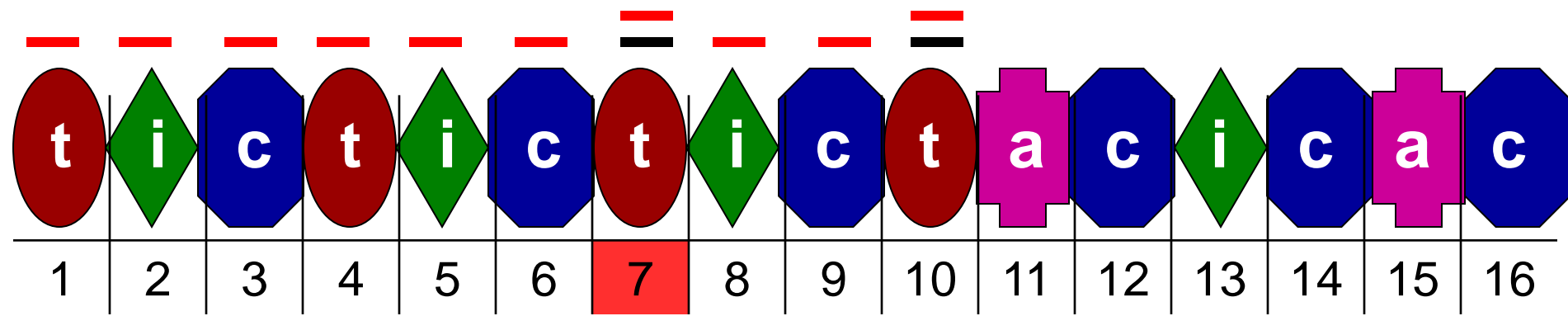
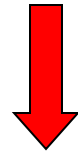
# KMP search



0	0	0	1	2	3
---	---	---	---	---	---

Consult  $OF(6)=3$  it tells how many positions backward from  $i$  the next comparison starts:  $k=i-OF(j)+1$

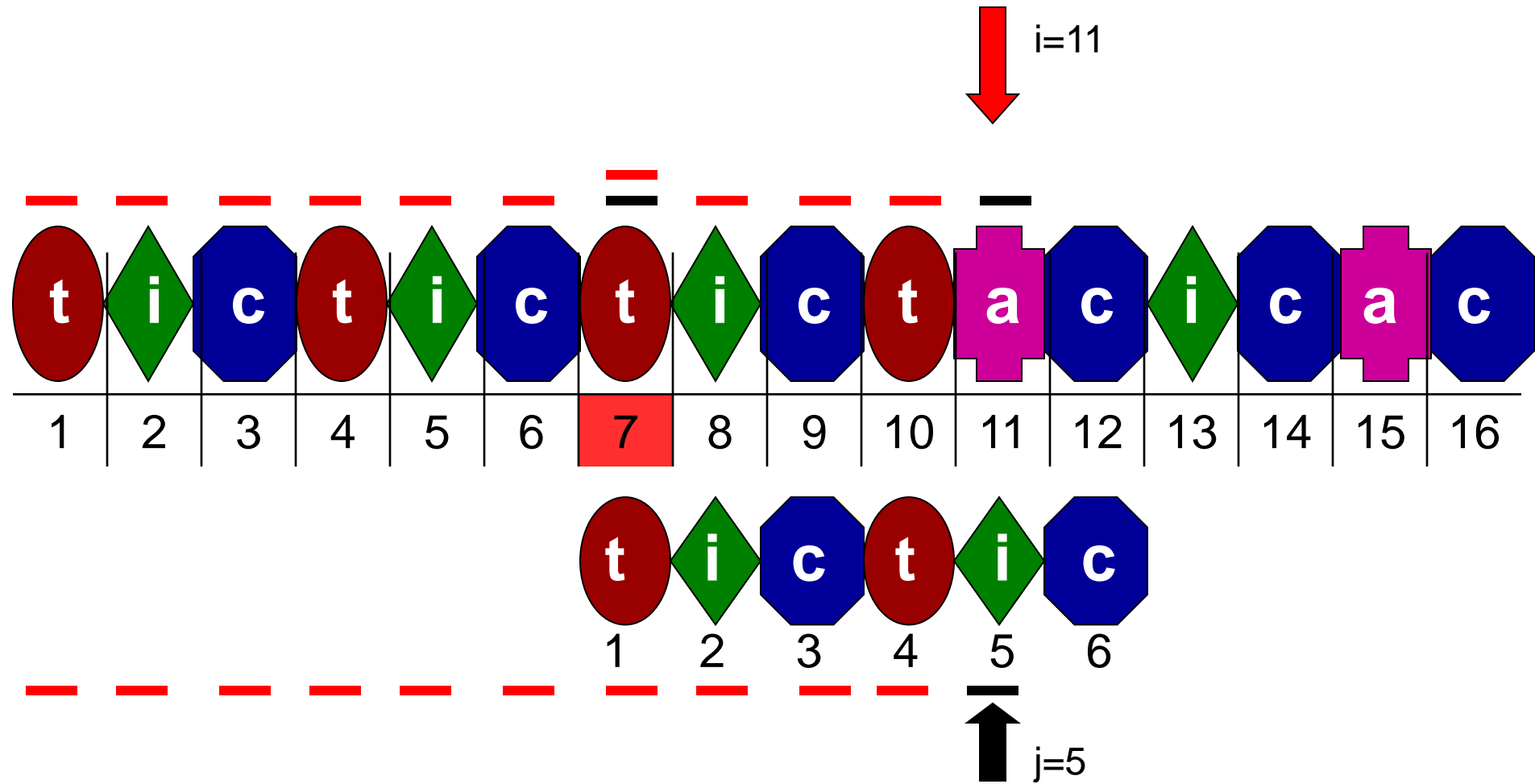
# KMP search



0	0	0	1	2	3
---	---	---	---	---	---

Continue comparing T[10] and P[4]

# KMP search

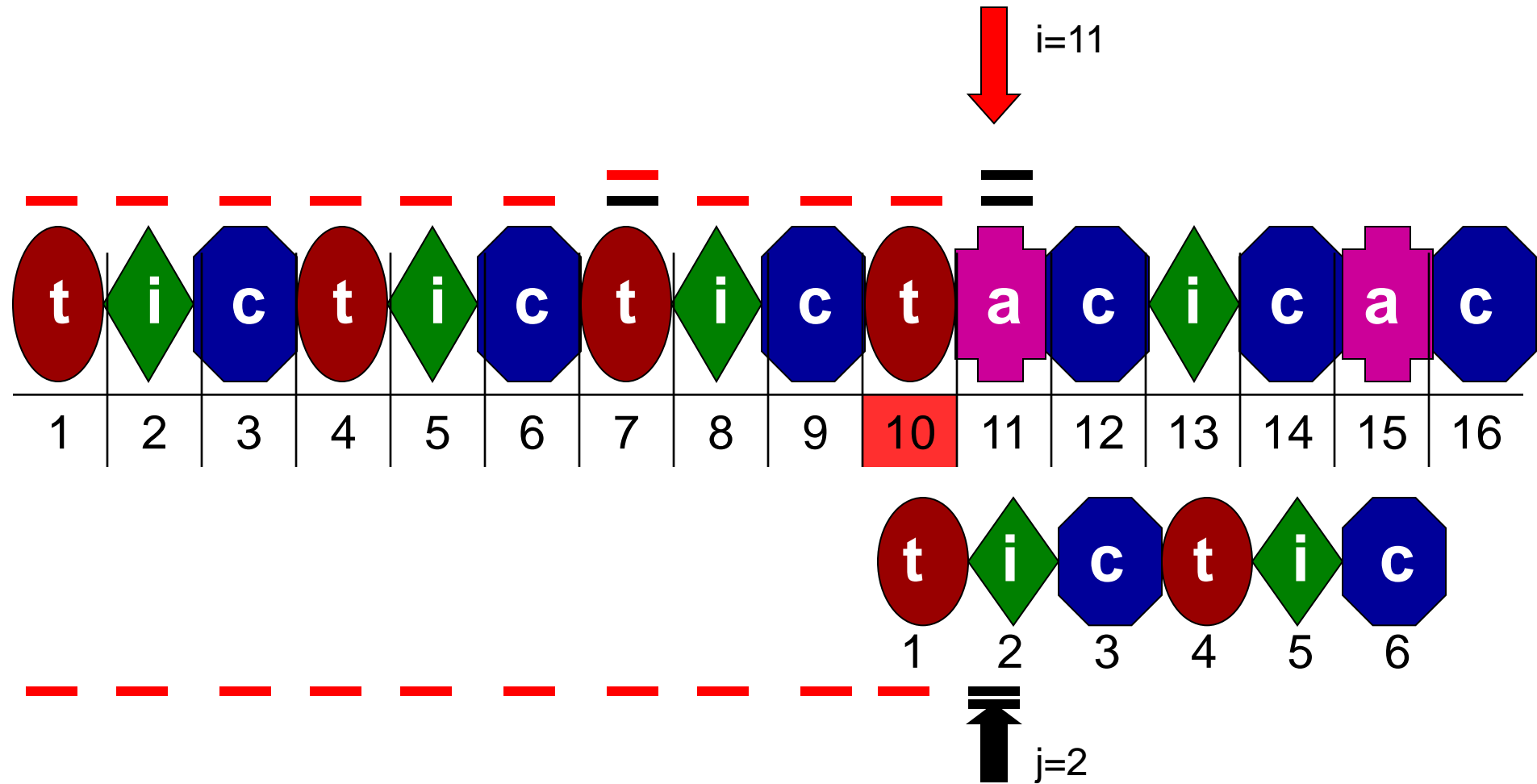


0	0	0	1	2	3
---	---	---	---	---	---

T[11] and P[5] do not match. Consult OF(4)=1. next potential match can start at  $i - OF(j) = 10$ , and the first character is already matched



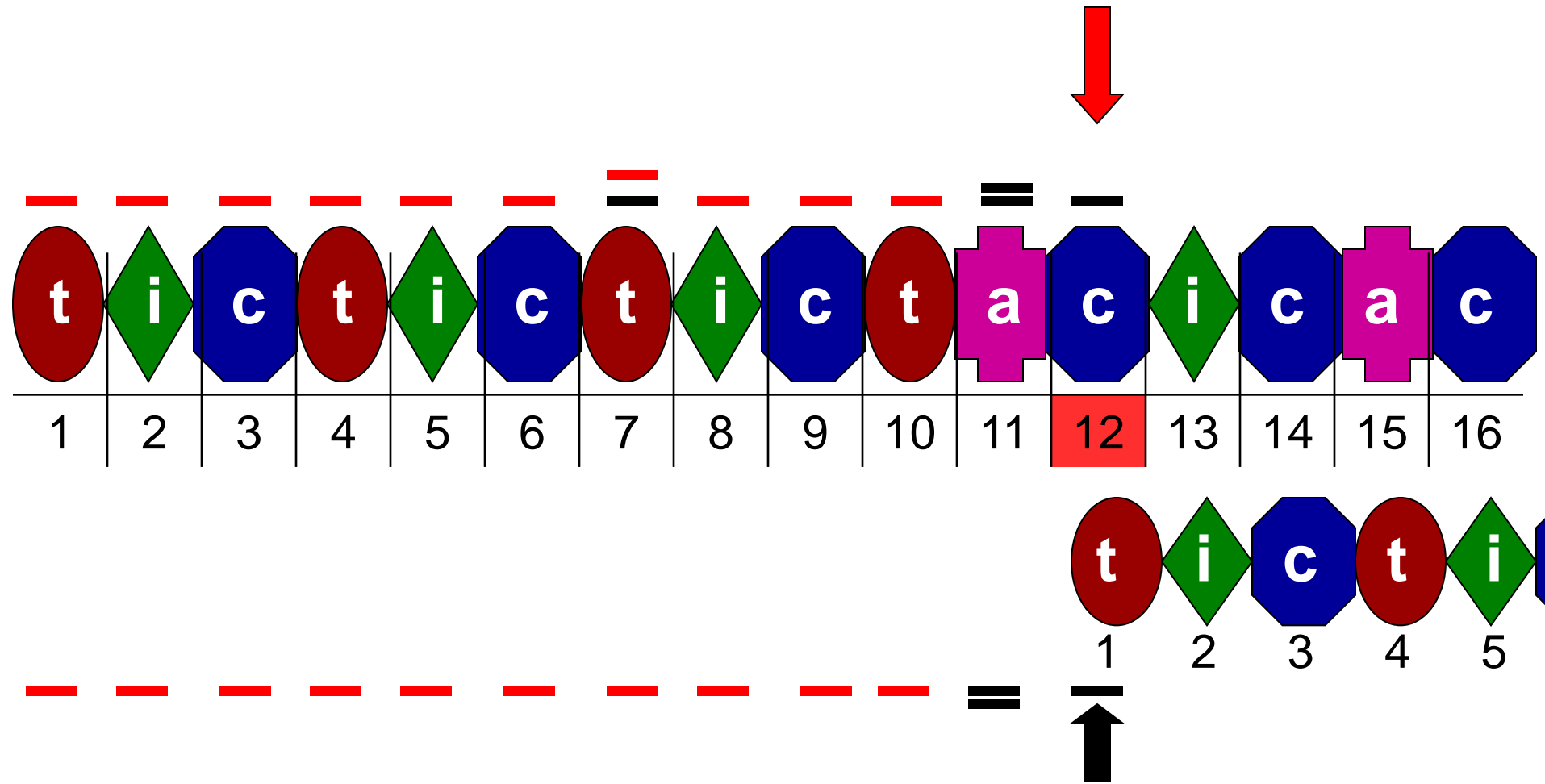
# KMP search



0	0	0	1	2	3
---	---	---	---	---	---

Here we only matched till the position  $j=2$ , the value  $OF(1)=0$ , therefore we are not shifting the start of the comparison backwards but starting from the next  $i=12$  etc...

# KMP search



0	0	0	1	2	3
---	---	---	---	---	---

If T would be larger, we continue in a similar manner, never accessing the characters of T more than twice

# KMP – from an intuition to the algorithm

We need 3 pointers (3 only for clarity, could work with 2):

- pointer  $i$  will point to the current character of text  $T$  of length  $N$
- pointer  $j$  will point to the current character of pattern  $P$  of length  $M$
- pointer  $k$  will point to the start of a current comparison in  $T$

in the beginning  $i=1, j=1, k=1$

$i:=1 \ j:=1 \ k:=1$

---

# KMP - from an intuition to the algorithm

if we have enough symbols in T to match P starting from position k, then we *continue* to compare the corresponding characters of P and T

```
i:=1 j:=1 k:=1  
while: N-k>=M
```

# KMP - from an intuition to the algorithm

we continue matching symbols of P  
while they match or until we reached  
the end of P

```
i:=1 j:=1 k:=1
while: N-k>=M
    while: j ≤ M and T[i]=P[j]
        i:=i+1
        j:=j+1
```

# KMP - from an intuition to the algorithm

If we reached the end of P, we found our match starting at position k of T

```
i:=1 j:=1 k:=1
while: N-k>=M
    while: j ≤ M and T[i]=P[j]
        i:=i+1
        j:=j+1
    if j>M then output k
```

# KMP - from an intuition to the algorithm

Now we need to find where to start the next comparison

if there was an overlap  $OL(j-1)$ , then

- set the start of a new comparison ( $k$ ) that many steps backwards from the current position in  $T$  as the value  $OL(j-1)$
- set  $j$  to the position  $OL(j-1)+1$  in  $T$  (we know that the previous characters match)
- $i$  remains unchanged, since now we are going to compare it with the symbol at a different position of  $P$

```
i:=1 j:=1 k:=1
```

```
while: N-k>=M
```

```
    while: j ≤ M and T[i]=P[j]
```

```
        i:=i+1
```

```
        j:=j+1
```

```
if j>M then output k
```

```
if  $OF(j-1)>0$ 
```

```
    k:=i- $OF(j-1)$ 
```

```
    j:= $OF(j-1)+1$ 
```

# KMP - from an intuition to the algorithm

if the value of an overlap function is zero (do not need to check backwards), then

- advance  $i$  to the next position
- set start of a comparison  $k$  to  $i$
- set  $j$  to 1

```
i:=1 j:=1 k:=1
while: N-k>=M
    while: j ≤ M and T[i]=P[j]
        i:=i+1
        j:=j+1
    if j>M then output k
    if OF(j-1)>0
        k:=i-OF(j-1)
        j:=OF(j-1)+1
    else
        i:=i+1 //only if we did not
        advance in T
        k:=i
        j:=1
```



# KMP - from an intuition to the algorithm

Note that in the **if/else** clauses:

$j = \text{OF}(j-1) + 1$  is the same as  
 $j = 1$ , for  $\text{OF}(j-1) = 0$

then we can shorten the pseudocode

```
i:=1 j:=1 k:=1
while: N-k>=M
    while: j ≤ M and T[i]=P[j]
        i:=i+1
        j:=j+1
    if j>M then output k
    if OF(j-1)>0
        k:=i-OF(j-1)
        j:=OF(j-1)+1
    else
        if i=k then
            i:=i+1
        k:=i
        j:=1
```

# KMP – algorithm complete

note that in if/else clauses:

$j = \text{OF}(j-1) + 1$  is the same as  
 $j = 1$ , for  $\text{OF}(j-1) = 0$

then we can shorten the pseudocode

```
i:=1 j:=1 k:=1
while: N-k>=M
    while: j ≤ M and T[i]=P[j]
        i:=i+1
        j:=j+1
    if j>M then output k
    j:=OF(j-1)+1
    if OF(j-1)>0
        k:=i-OF(j-1)
    else
        if i=k then
            i:=i+1
        k:=i
```

# KMP – the final pseudocode

**algorithm KMP** ( $T$  of length  $N$ ,  $P$  of length  $M$ )

$i:=1$   $j:=1$   $k:=1$

**while:**  $N-k \geq M$

**while:**  $j \leq M$  and  $T[i]=P[j]$

$i:=i+1$

$j:=j+1$

**if**  $j > M$  **then output**  $k$

$j:=OF(j-1)+1$

**if**  $OF(j-1) > 0$  **then**

$k:=i-OF(j-1)$

**else**

**if**  $i=k$  **then**

$i:=i+1$

$k:=i$

# A KMP code for the 0-base array (C-code)

```
while ((N-k)>=M)
{
    while( j <M && T[i]==P[j])
    {
        i++;
        j++;
    }
    if (j==M)
        printf ("Occurence at pos %d\n",k);

    stepBack=0;
    if (j>1)
    {
        stepBack=OF[j-1];
        j=OF[j-1]+1;
    }
    if (stepBack>0)
        k=i-stepBack;
    else
    {
        if(i==k)
            i++;

        k=i;
    }
}
```

# KMP algorithm time complexity

- The number of character comparisons in KMP algorithm is at most  $2N$ 
  - Divide the algorithm into compare/shift phases, where a single phase consists of the comparisons done between 2 successive shifts. During 2 consecutive shifts, at most 2 comparisons are done for each character of  $T$ . Since pattern is never shifted left, the total number of character comparisons is bounded by  $N+s$ , where  $s$  is the total number of shifts. But  $s < N$ , since after  $N$  shifts the right end of  $P$  is certainly to the right of the right end of  $T$ , so the total number of comparisons done is bounded by  $2N$

# A worst-case example – iterations 1,2

1	1	1	1	1					
a	a	a	a	b	a	a	a	a	a
a	a	a	a	a					

We have aligned pattern P, by performing so far 1 character comparison for each of 5 characters of P

Now we need to restart the comparison from the position 2 of T

1	1	1	1	2					
a	a	a	a	b	a	a	a	a	a
	a	a	a	a	a				

# A worst-case example – iteration 3

1	1	1	1	2					
a	a	a	a	b	a	a	a	a	a
	a	a	a	a	a				

We have compared character b of T already 2 times  
Next we start by aligning pattern starting at position 3 of T

1	1	1	1	3					
a	a	a	a	b	a	a	a	a	a
	a	a	a	a	a				





# A worst-case example – iteration 5

1	1	1	1	5					
a	a	a	a	b	a	a	a	a	a

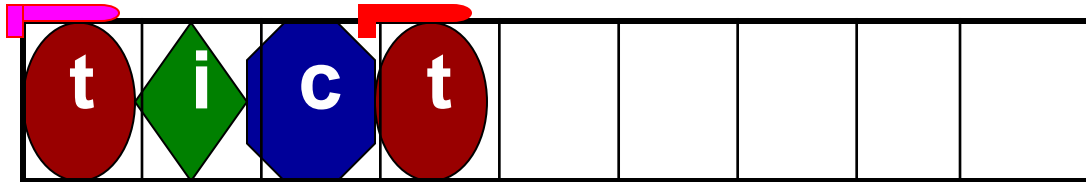
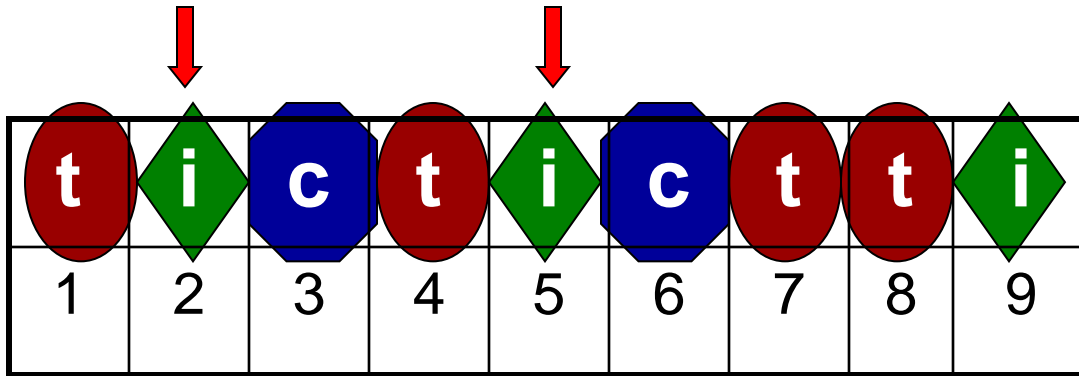
  

a	a	a	a	a

For now, we have compared character b of T 5 times (as the length of the pattern), but during this comparison we have shifted the left end of P 5 positions forward. Since we did not compare anymore any character to the left from b, we did in total not more than  $5 \cdot 2$  comparisons in order to process the 5 first characters of T.

This is true in general: the total number of character comparisons in KMP is bounded by  $2N$

# How to compute the OF function



1	2	3	4	5	6	7	8	9
0	0	0	1					

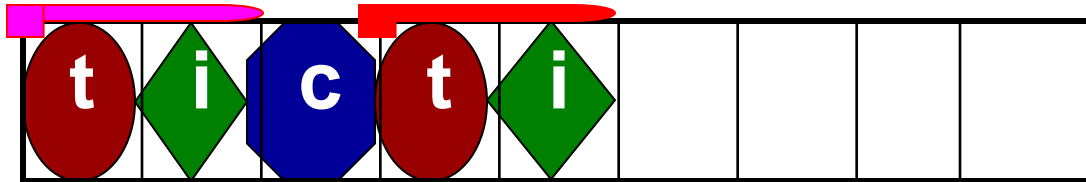
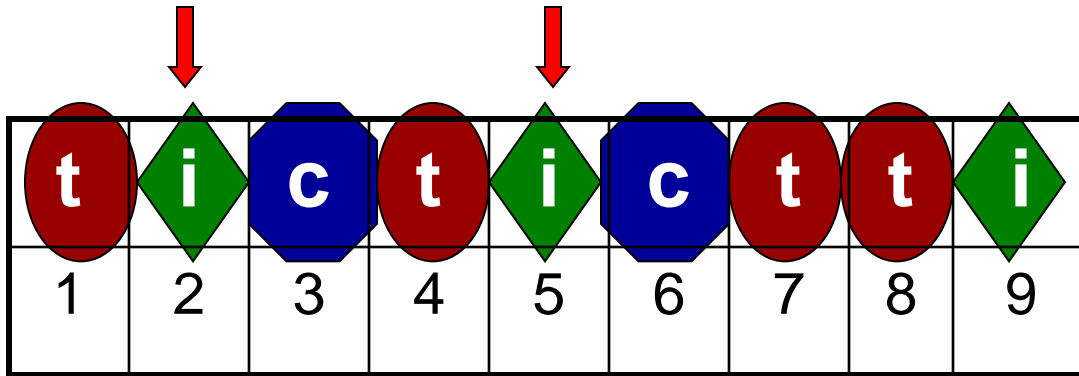
The easy case:

if we have  $OF(j-1)$ , and the characters

$P[j]$  and  $P[OF(j-1)+1]$  match

Then we just increase by 1  
 $OF(j) = OF(j-1) + 1$

# How to compute the OF function



1	2	3	4	5	6	7	8	9
0	0	0	1	2				

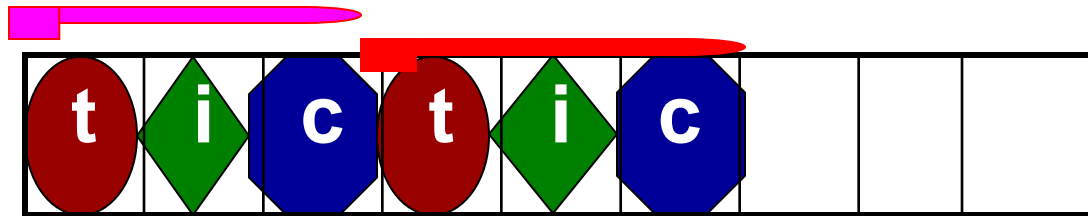
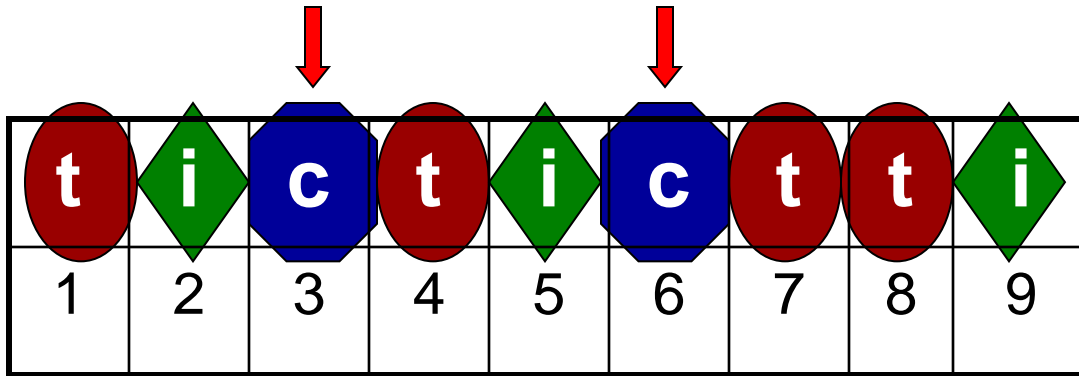
The easy case:

if we have  $OF(j-1)$ , and the characters

$P[j]$  and  $P[OF(j-1)+1]$  match

Then we just increase  
 $OF(j)=OF(j-1)+1$

# How to compute the OF function



1	2	3	4	5	6	7	8	9
0	0	0	1	2	3			

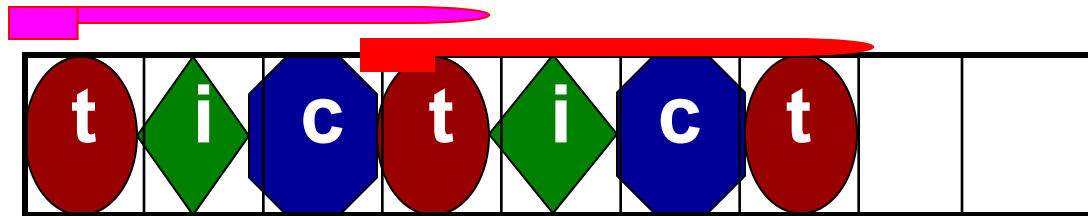
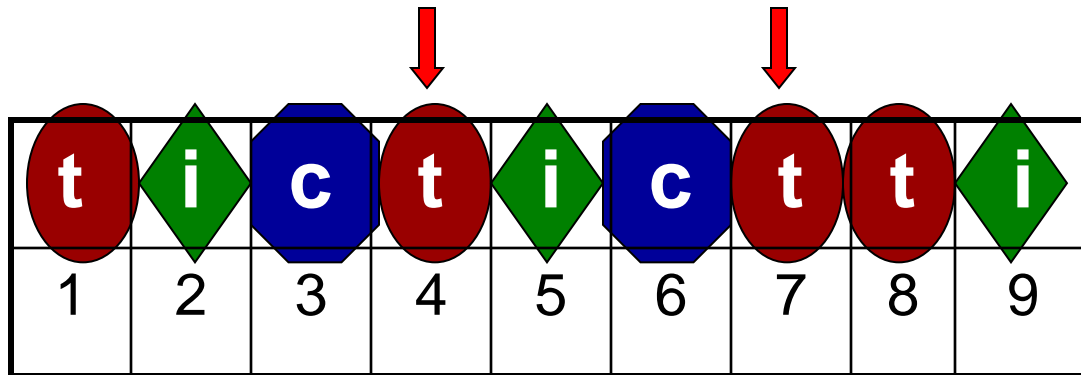
The easy case:

if we have  $OF(j-1)$ , and the characters

$P[j]$  and  $P[OF(j-1)+1]$  match

Then we just increase  
 $OF(j)=OF(j-1)+1$

# How to compute the OF function



1	2	3	4	5	6	7	8	9
0	0	0	1	2	3	4		

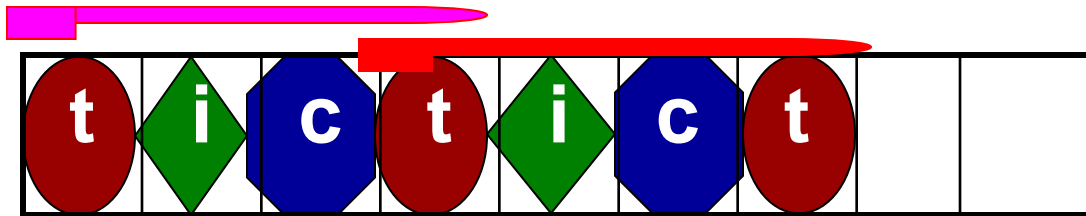
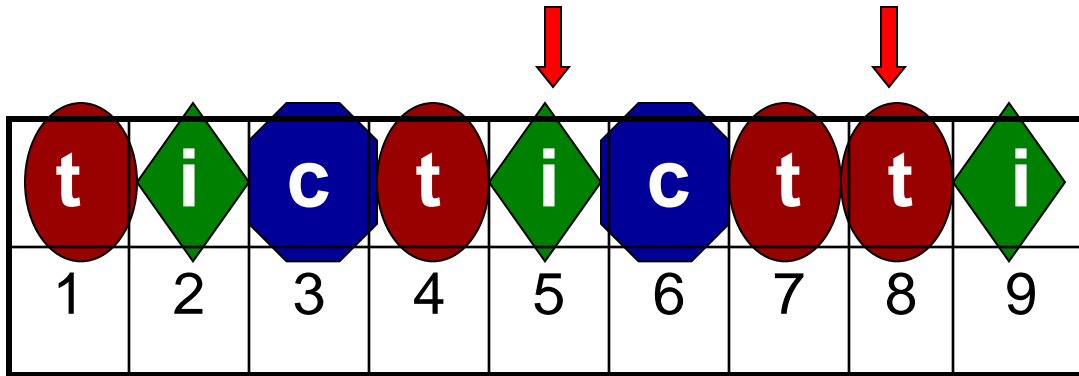
The easy case:

if we have  $OF(j-1)$ , and the characters

$P[j]$  and  $P[OF(j-1)+1]$  match

Then we just increase  
 $OF(j)=OF(j-1)+1$

# How to compute the OF function



1	2	3	4	5	6	7	8	9
0	0	0	1	2	3	4		

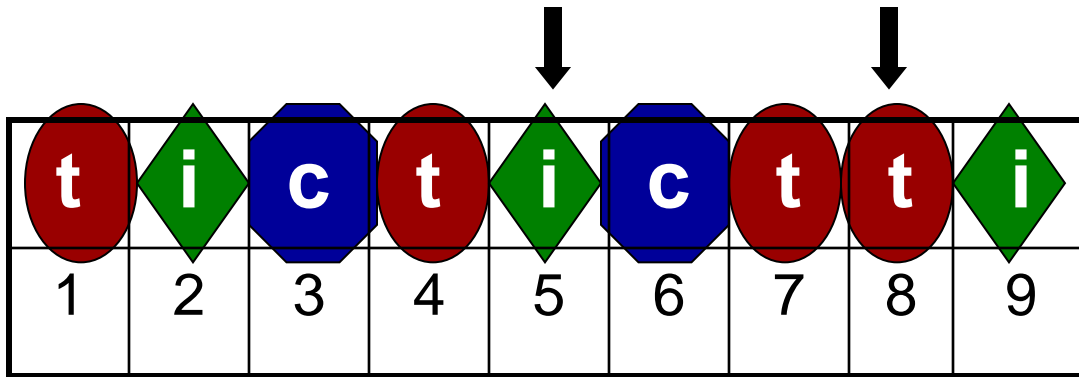
The *general* case:

If the characters

$P[j]$  and  $P[OF(j-1)+1]$  do not match

where do we find  $OF[j]$ ?

# How to compute the OF function



The general case:

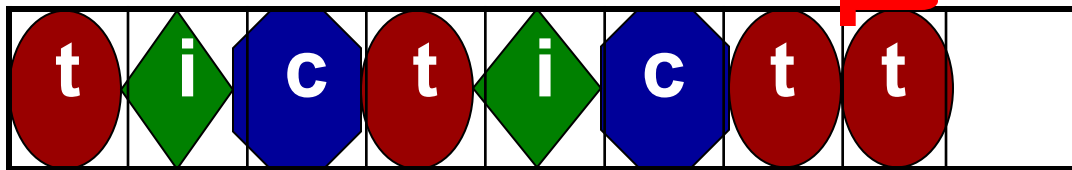
If the characters

$P[j]$  and  $P[OF(j-1)+1]$  do not match

then  $OF(j)$  is less than  $OF(j-1)$

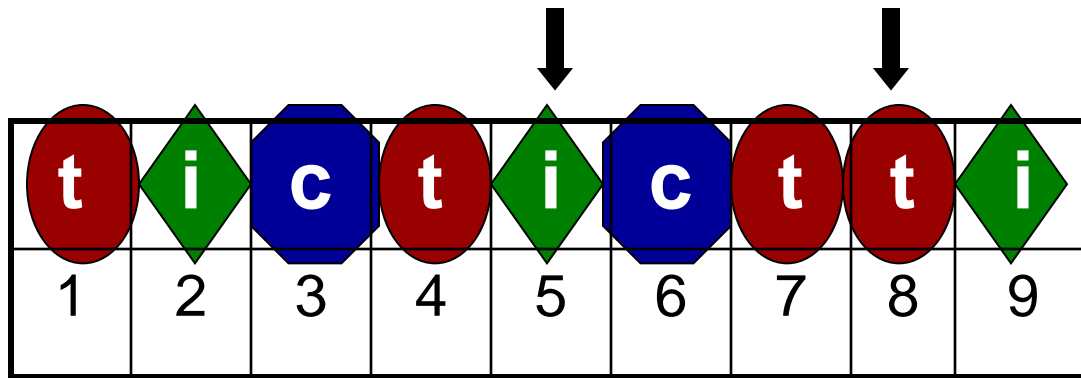
We look at  $v = OF(j-1)$  and check again the next character

$P[OF(v)+1]$



1	2	3	4	5	6	7	8	9
0	0	0	1	2	3	4		

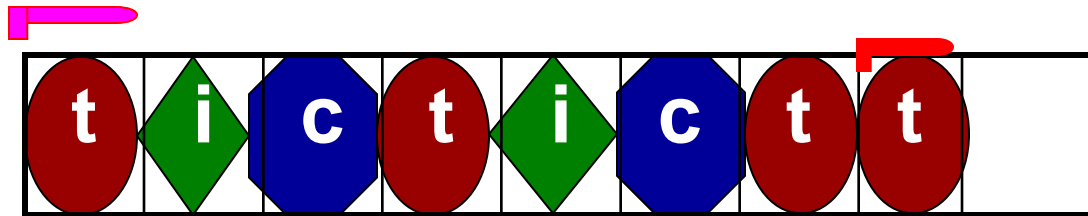
# How to compute the OF function



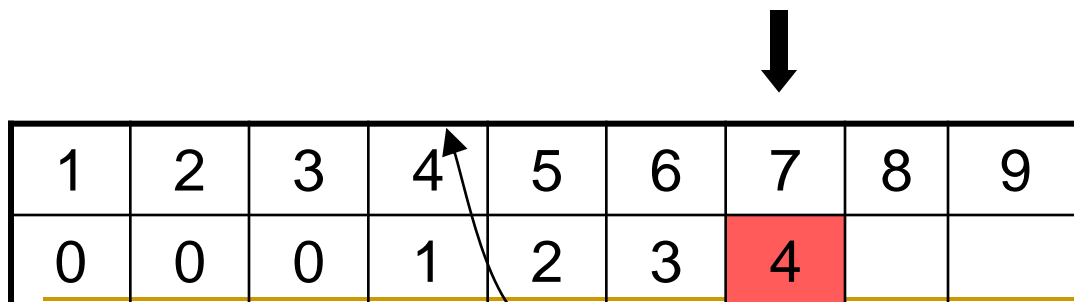
The general case:

If the characters

$P[j]$  and  $P[OF(j-1)+1]$  do not match



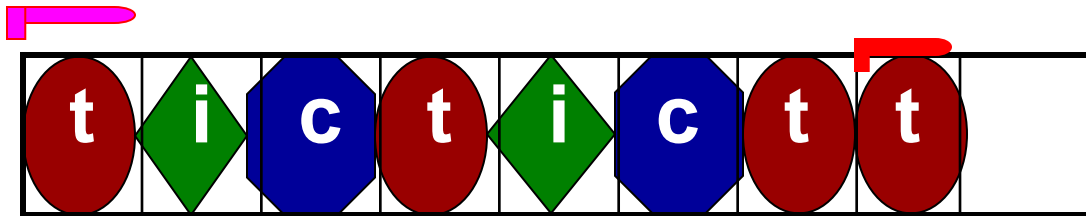
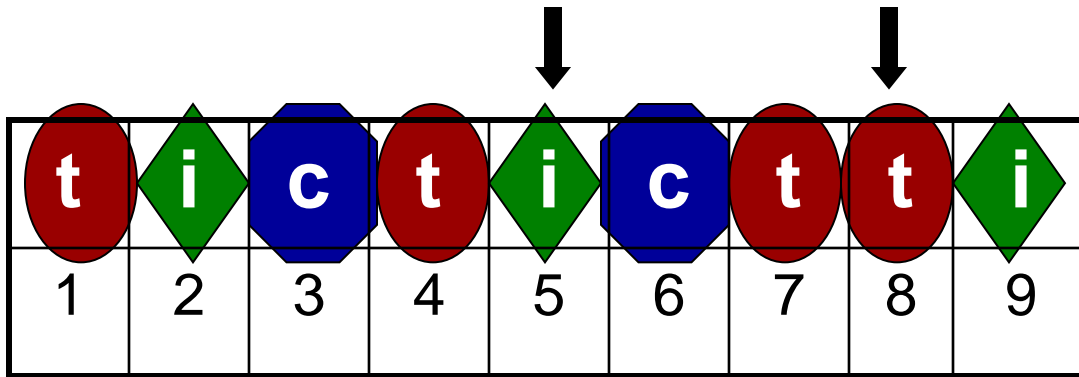
we look at  $v=OF(j-1)$  and check again the next character  $P[OF(v)+1]$



The pointer is bouncing through the entire OF table until it finds the symbol matching the current symbol after the next assignment of  $v=OF(v)$



# How to compute the OF function



1	2	3	4	5	6	7	8	9
0	0	0	1	2	3	4		

The general case:

If the characters

$P[j]$  and  $P[OF(j-1)+1]$  do not match

then  $OF(j)$  is less than  $OF(j-1)$

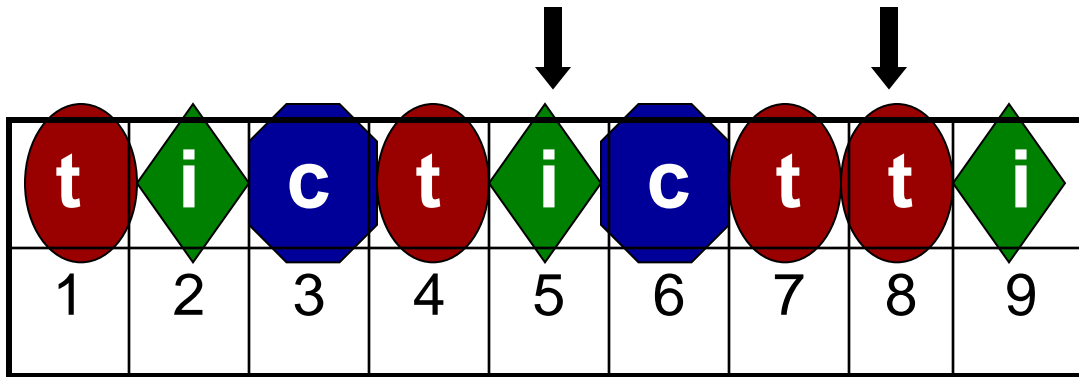
We look at  $v=OF(j-1)$  and check again the next character

The pointer is bouncing through the entire OF table until it finds the symbol matching the current symbol after the next assignment of  $v=OF(v)$

$P[2] \neq P[8]$

$v=OF(4)$

# How to compute the OF function

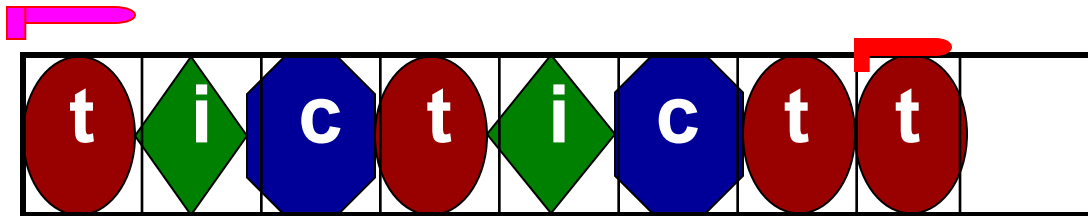


The general case:

$$v = \text{OF}(4)$$

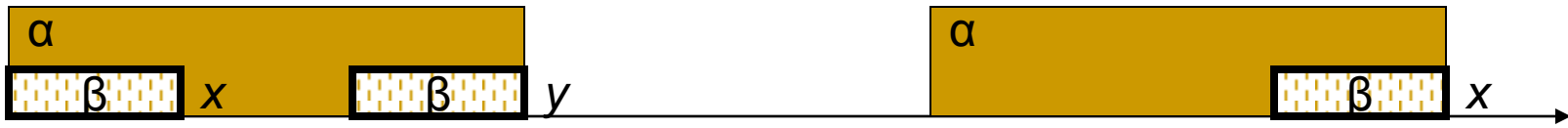
$P[1] = P[8]$ , thus

$$\text{OF}(8) = \text{OF}(1) + 1 = 1$$



1	2	3	4	5	6	7	8	9
0	0	0	1	2	3	4		

# Why do we compute the OF value this way?



We know that since we could not extend suffix  $\alpha$ , so there is a smaller suffix,  $\beta$ , which starts somewhere inside  $\alpha$ .

What is the next smaller overlap for all suffixes starting inside  $\alpha$ ?

The same as for all suffixes inside the prefix of length  $|\alpha|$

Thus, if we check the OF value for the position  $|\alpha|$ , we see the next smaller maximal overlap

We check if this is a desired maximal overlap by checking the next character after the prefix of size  $|\beta|$

If this character is  $x$ , we are done

If not, we continue by the same logic

# Example

<b>t</b>	<b>i</b>	<b>c</b>	<b>t</b>	<b>i</b>	<b>c</b>	<b>t</b>	<b>t</b>	<b>i</b>
1	2	3	4	5	6	7	8	9

<b>t</b>	<b>i</b>	<b>c</b>	<b>t</b>	<b>i</b>	<b>c</b>	<b>t</b>	<b>t</b>	
1	2	3	4	5	6	7	8	9

1	2	3	4	5	6	7	8	9
0	0	0	<b>1</b>	2	3	4		

We know that the substring *tictict* ending at position 7 had suffix *tict* which is overlapping with the prefix *tict* of the pattern

We also know that we cannot extend this overlap since  $P[8]$  and  $P[5]$  do not match

Now we want to check what overlap had the prefix *tict* with the prefix of the entire pattern, since the suffix start for a new overlap is somewhere inside *tict*

We look at position 4 in OF table and find that the next overlap for substring of length 4 is of length 1

# Example

<b>t</b>	<b>i</b>	<b>c</b>	<b>t</b>	<b>i</b>	<b>c</b>	<b>t</b>	<b>t</b>	<b>i</b>
1	2	3	4	5	6	7	8	9

<b>t</b>	<b>i</b>	<b>c</b>	<b>t</b>	<b>i</b>	<b>c</b>	<b>t</b>	<b>t</b>	
1	2	3	4	5	6	7	8	9

1	2	3	4	5	6	7	8	9
<b>0</b>	0	0	1	2	3	4		

We check if  $P[1+1]$  matches  $P[8]$

They do not

We repeat and by the same logic we are going to the entry 1 of the OF table, and find that there is no overlap for this value:  $OF[1]=0$

So we check if

$P[0+1]$  matches  $P[8]$

They do, so the  $OF[8]=OF[1]+1=1$

# A more complex example of the OL computation

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
	c	a	t	c	a	p	c	a	t	c	a	r	c	a	t	c	a	p	c	a	t	c	a	t
OL	0	0	0	1	2	0	1	2	3	4	5	0	1	2	3	4	5	6	7	8	9	1	1	?

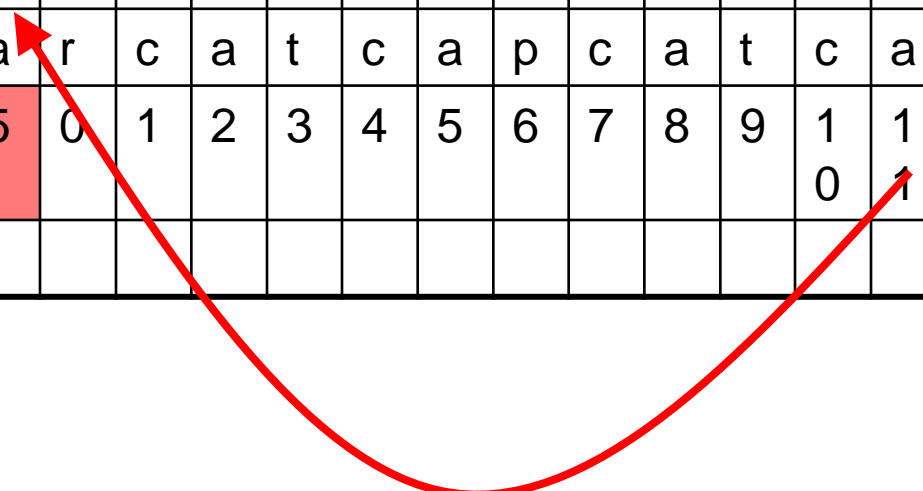
We know that  $OL(23)=11$

This means that the sequence of the first 11 characters of P is the same as that of the last 11 characters of P[1....23]

However, the character  $P[11+1]=r$  does not match the character  $P[23+1]=t$

# A more complex example of the OL computation

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
	c	a	t	c	a	p	c	a	t	c	a	r	c	a	t	c	a	p	c	a	t	c	a	t
OL	0	0	0	1	2	0	1	2	3	4	5	0	1	2	3	4	5	6	7	8	9	1	1	?



The maximum possible overlap is less than 11

The next maximum possible overlap can be found if we look at position 11 of the OF table and see what overlap this substring had

The substring  $P[1\dots 11]$  has a maximum overlap of length 5

# A more complex example of the OL computation

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
	c	a	t	c	a	p	c	a	t	c	a	r	c	a	t	c	a	p	c	a	t	c	a	t
OL	0	0	0	1	2	0	1	2	3	4	5	0	1	2	3	4	5	6	7	8	9	1	1	?

Let us check if this value is also the maximum overlap for the substring  $P[1\dots 24]$

For this we check the character next to  $P[5]$ , which is p, and it does not match our t

Therefore, the overlap we are looking for is less than 5

---



# A more complex example of the OL computation



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
	c	a	t	c	a	p	c	a	t	c	a	r	c	a	t	c	a	p	c	a	t	c	a	t
OL	0	0	0	1	2	0	1	2	3	4	5	0	1	2	3	4	5	6	7	8	9	1	1	3

We check the next possible value by considering the overlap value for the substring  $P[1\dots 5]$

This value is 2. Is this value of an overlap good for  $P[1\dots 24]$ ?


We check  $P[2+1]=t$ , and  $P[24]=t$

Thus, the overlap for the substring  $P[1\dots 24]$  is  $2+1=3$

# Practice jumps on the following pattern

- aaahamaaahamamaaahamaaaa

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
	a	a	a	h	a	m	a	a	a	h	a	m	a	m	a	a	a	h	a	m	a	a	a	a
O L	0	1	2	0	1	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	8	9	?









# Overlap function - pseudocode

**algorithm computeOverlapFunction** (pattern  $P$  of length  $M$ )

$OF[1]=0$

**for**  $k:=1$  **to**  $M-1$

$c:=P[k+1]$  // current character of  $P$

$v:=OF[k]$

**while:**  $P[v+1] \neq c$  **and**  $v \neq 0$

$v:=OF[v]$

**if**  $P[v+1]=c$

$OF[k+1]:=v+1$

**else**

$OF[k+1]:=0$

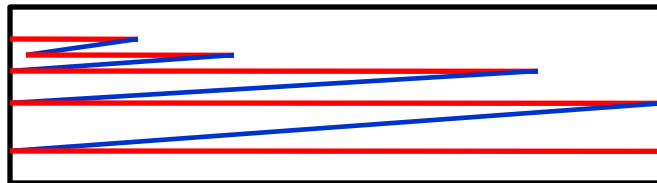
**return**  $OF$  table

# Overlap function: time complexity

The computation of  $OF$  is performed in time  $O(M)$  since:

- the total complexity is proportional to the total number of times the value of  $v$  is changed
- this value is increasing by one (or remains zero) in the *for* loop, and in total, during the entire algorithm, it is increasing not more than by  $M$  units
- in addition, the value of  $v$  is decreasing inside the *while* loop, but since  $v$  is never less than zero, the total number of units by which it is decreasing can not be more than the number it has been increasing, therefore it is bounded by  $M$  too.

The time is therefore less than  $2M$ :  $O(M)$



If we sum up the length of all the red lines (increasing value of  $v$ ), the result will be  $\leq M$ . Therefore, the total length of blue lines (decreasing value of  $v$ ) cannot be more than  $M$  in total

---

# Overlap function table

- Is called in the modern literature the *border array*



# Overlap Function - again

1	2	3	4	5	6
<b>a</b>	<b>a</b>	<b>b</b>	<b>a</b>	<b>a</b>	<b>a</b>

OF

0	1	0	1	2	2
---	---	---	---	---	---

The *OF* values tell where to position the start of the next comparison

They also tell which character to compare in *P* and whether to advance or not the pointer in *T*

For example, if mismatch occurred at pattern position  $j=5$ , from  $OF(5-1)=1$  the start  $k$  is 1 position backwards from a current position  $i$  in *T*, and we compare the same character in *T* with the character  $OF(5-1)+1=2$  in *P*, since we know that the first 1 character matches *T* starting from  $k$

pos	OF		
1	0	Advance in T	Compare P[1]
2	1	Stay in T	Compare P[1]
3	0	Stay in T	Compare P[2]
4	1	Stay in T	Compare P[1]
5	2	Stay in T	Compare P[2]
6	2	Stay in T	Compare P[3]
		Advance in T	Compare P[3]

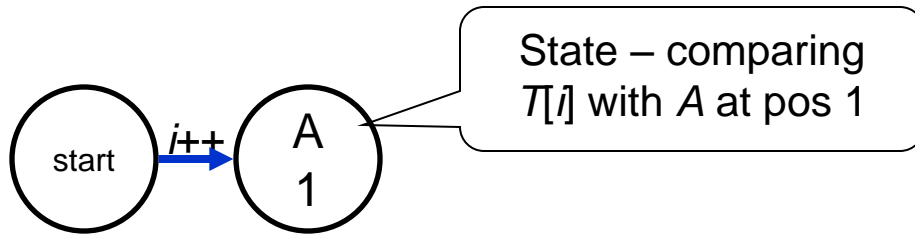
---

# Finite state automaton

- *FSA* is a model of behavior composed of a finite number of *states*, *transitions* between those states, and *actions*.
  - It is similar to a "flow graph" where we can inspect the way in which the logic runs when certain conditions are met.
-

# KMP pattern matching automaton

1	2	3	4	5	6
<b>a</b>	<b>a</b>	<b>b</b>	<b>a</b>	<b>a</b>	<b>a</b>

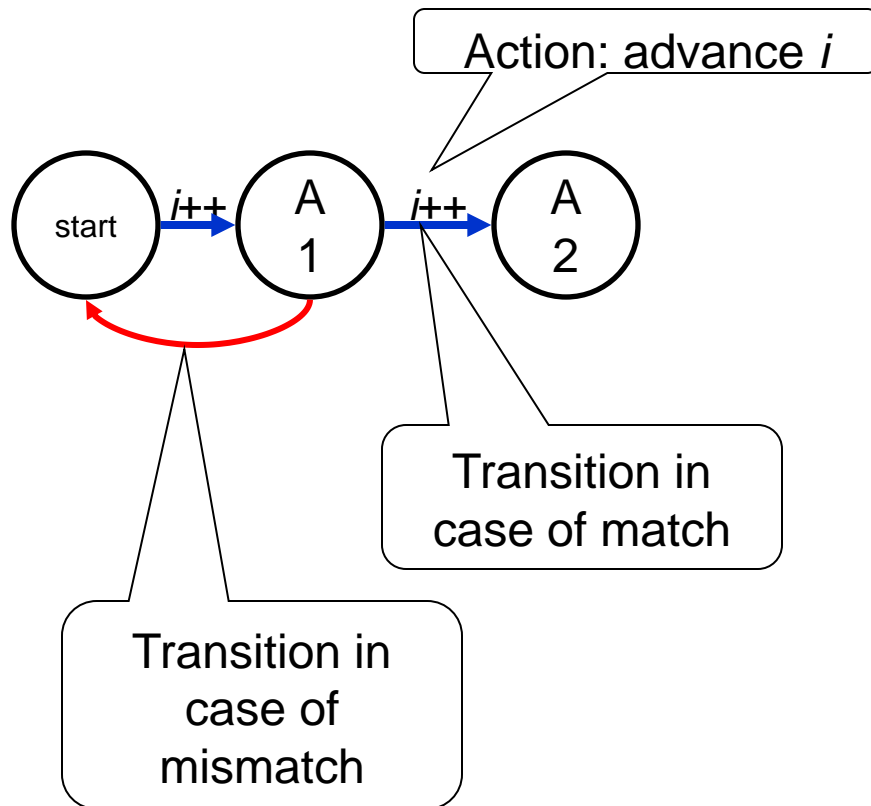


# KMP pattern matching automaton

1	2	3	4	5	6
<b>a</b>	<b>a</b>	<b>b</b>	<b>a</b>	<b>a</b>	<b>a</b>

OF

0	1	0	1	2	2
---	---	---	---	---	---



Where the transition in case of failure is directed, is determined by the value of an overlap function

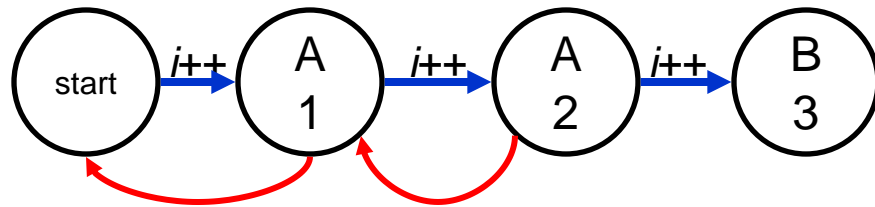
That is why OL function is called also *a failure function*

# KMP pattern matching automaton

1	2	3	4	5	6
<b>a</b>	<b>a</b>	<b>b</b>	<b>a</b>	<b>a</b>	<b>a</b>

OF

0	1	0	1	2	2
---	---	---	---	---	---

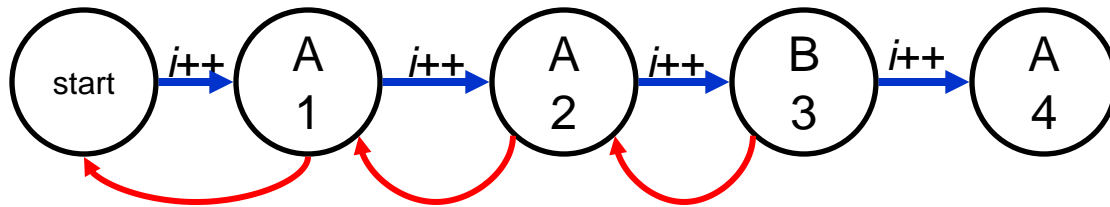


# KMP pattern matching automaton

1	2	3	4	5	6
<b>a</b>	<b>a</b>	<b>b</b>	<b>a</b>	<b>a</b>	<b>a</b>

OF

0	1	0	1	2	2
---	---	---	---	---	---

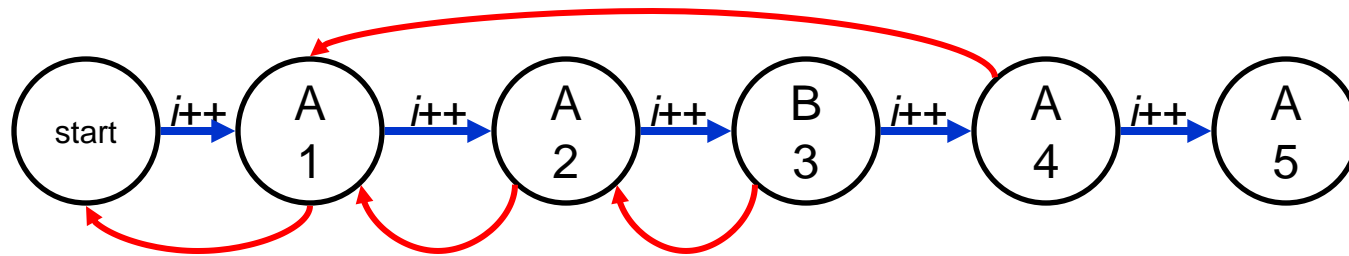


# KMP pattern matching automaton

1	2	3	4	5	6
<b>a</b>	<b>a</b>	<b>b</b>	<b>a</b>	<b>a</b>	<b>a</b>

OF

0	1	0	1	2	2
---	---	---	---	---	---

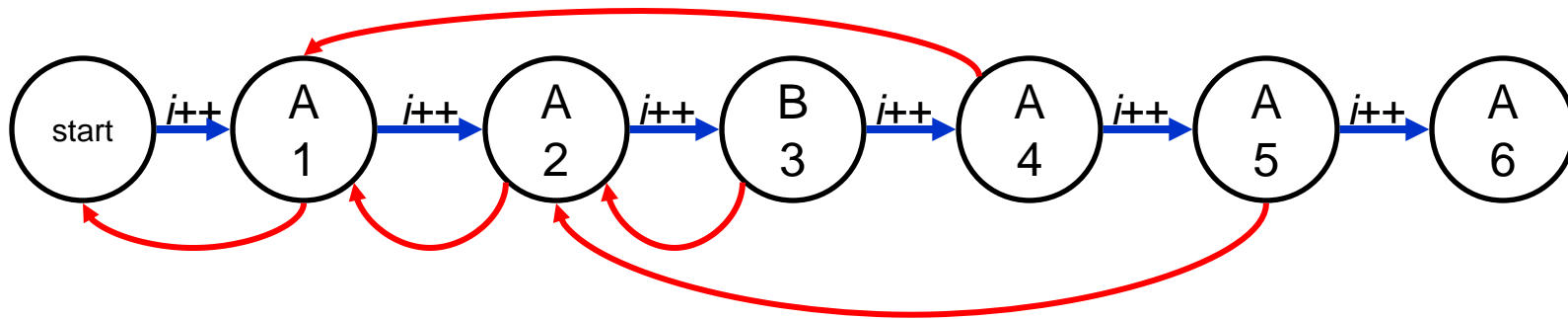


# KMP pattern matching automaton

1	2	3	4	5	6
<b>a</b>	<b>a</b>	<b>b</b>	<b>a</b>	<b>a</b>	<b>a</b>

OF

0	1	0	1	2	2
---	---	---	---	---	---



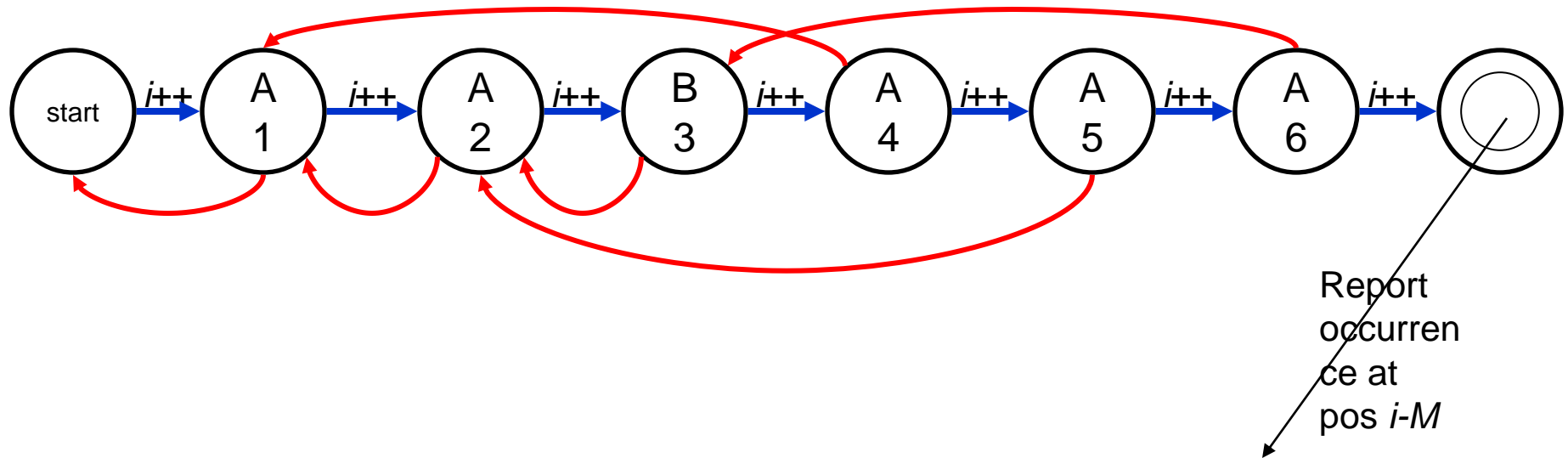


# KMP pattern matching automaton

1	2	3	4	5	6
<b>a</b>	<b>a</b>	<b>b</b>	<b>a</b>	<b>a</b>	<b>a</b>

OF

0	1	0	1	2	2
---	---	---	---	---	---

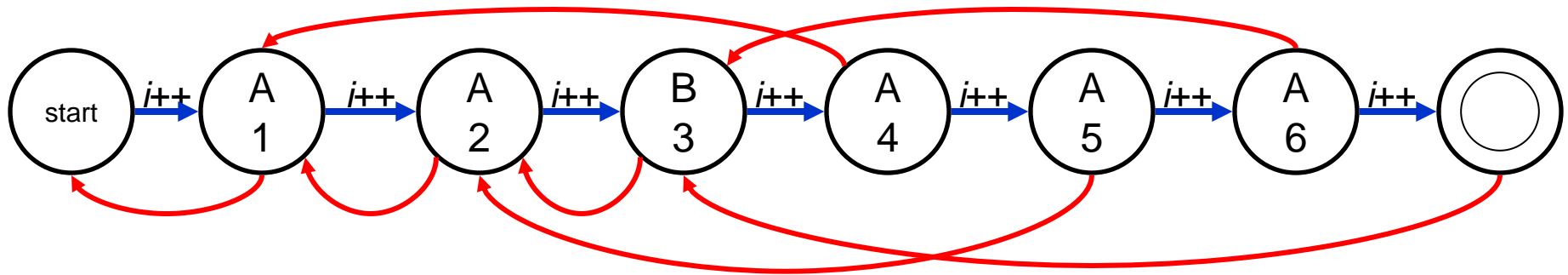


# KMP pattern matching automaton

1	2	3	4	5	6
<b>a</b>	<b>a</b>	<b>b</b>	<b>a</b>	<b>a</b>	<b>a</b>

OF

0	1	0	1	2	2
---	---	---	---	---	---

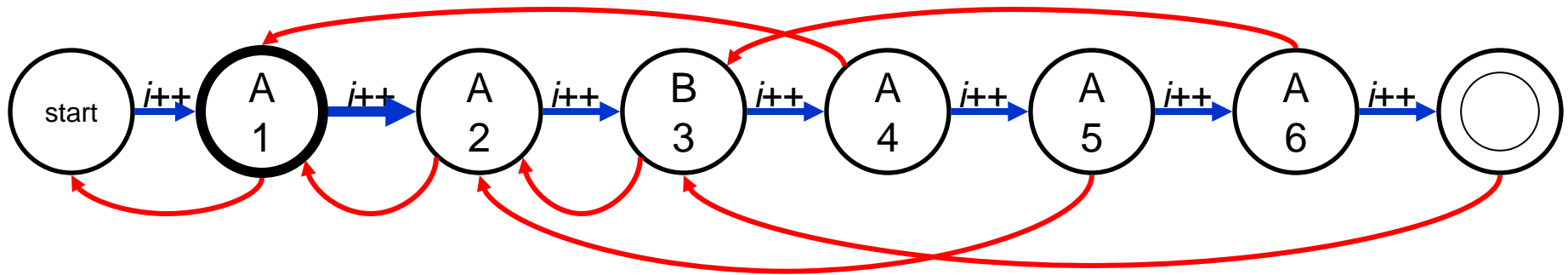


# KMP pattern matching automaton

1	2	3	4	5	6
<b>a</b>	<b>a</b>	<b>b</b>	<b>a</b>	<b>a</b>	<b>a</b>

OF

0	1	0	1	2	2
---	---	---	---	---	---



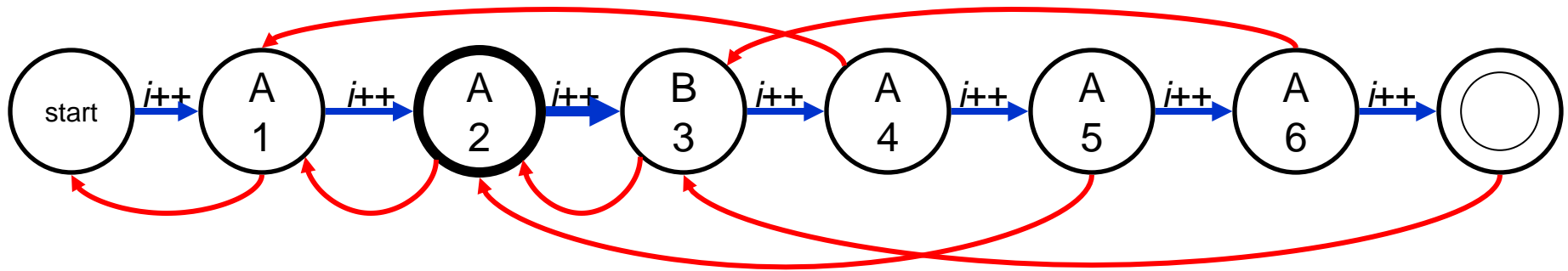
Example: streaming text  $T = \underline{a}aabaaaaabaaaa$  through the automaton

# KMP pattern matching automaton

1	2	3	4	5	6
<b>a</b>	<b>a</b>	<b>b</b>	<b>a</b>	<b>a</b>	<b>a</b>

OF

0	1	0	1	2	2
---	---	---	---	---	---



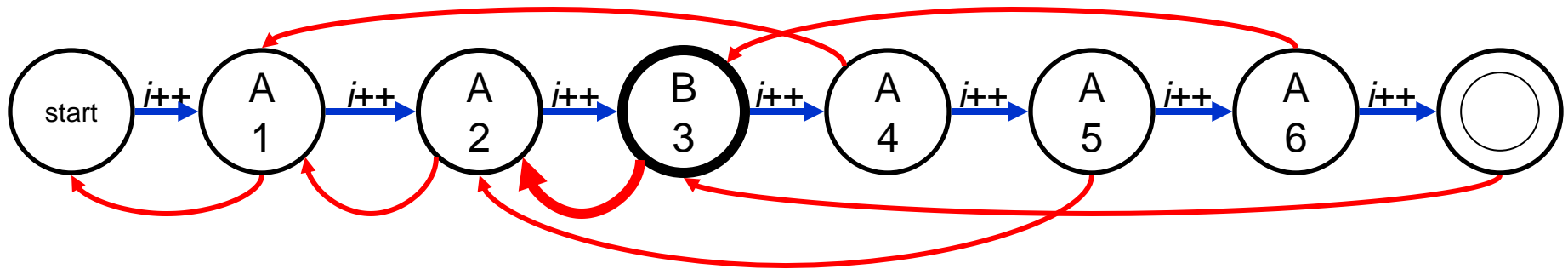
Example: streaming text  $T = a**a**abaaaaabaaaa$  through the automaton

# KMP pattern matching automaton

1	2	3	4	5	6
<b>a</b>	<b>a</b>	<b>b</b>	<b>a</b>	<b>a</b>	<b>a</b>

OF

0	1	0	1	2	2
---	---	---	---	---	---



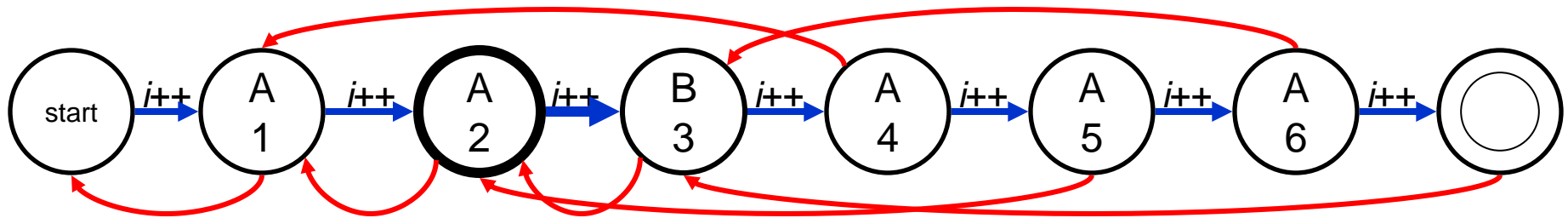
Example: streaming text  $T=aa**a**baaaaabaaaa$  through the automaton

# KMP pattern matching automaton

1	2	3	4	5	6
<b>a</b>	<b>a</b>	<b>b</b>	<b>a</b>	<b>a</b>	<b>a</b>

OF

0	1	0	1	2	2
---	---	---	---	---	---



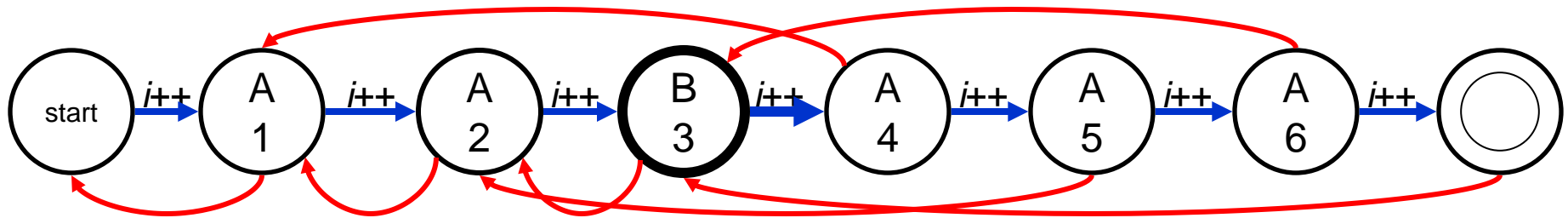
Example: streaming text  $T=aa**a**baaaaabaaaa$  through the automaton

# KMP pattern matching automaton

1	2	3	4	5	6
<b>a</b>	<b>a</b>	<b>b</b>	<b>a</b>	<b>a</b>	<b>a</b>

OF

0	1	0	1	2	2
---	---	---	---	---	---



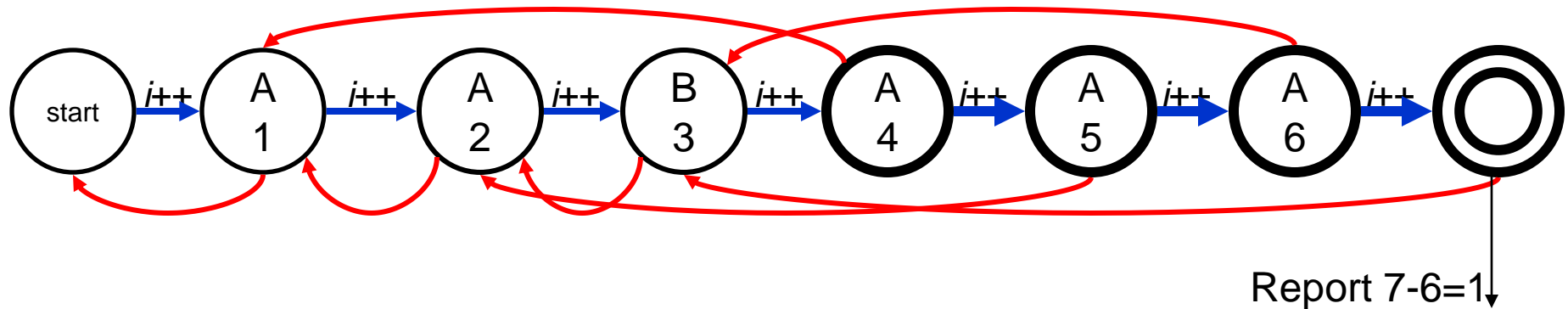
Example: streaming text  $T=aaabaaaaabaaaa$  through the automaton

# KMP pattern matching automaton

1	2	3	4	5	6
<b>a</b>	<b>a</b>	<b>b</b>	<b>a</b>	<b>a</b>	<b>a</b>

OF

0	1	0	1	2	2
---	---	---	---	---	---



Example: streaming text  $T=aaabaaaabaaaa$  through the automaton

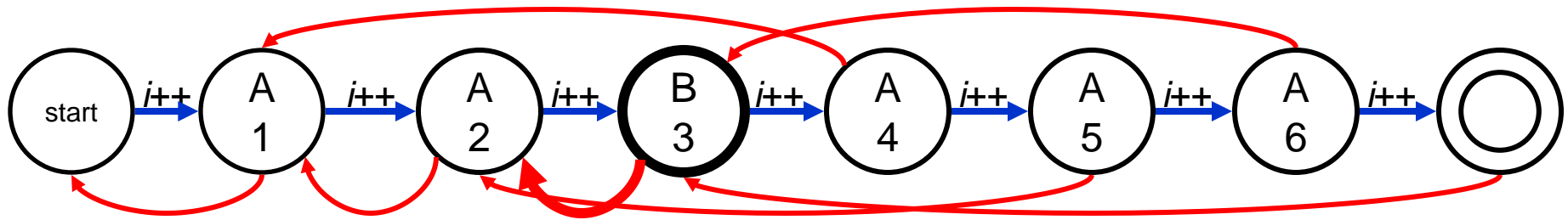


# KMP pattern matching automaton

1	2	3	4	5	6
<b>a</b>	<b>a</b>	<b>b</b>	<b>a</b>	<b>a</b>	<b>a</b>

OF

0	1	0	1	2	2
---	---	---	---	---	---



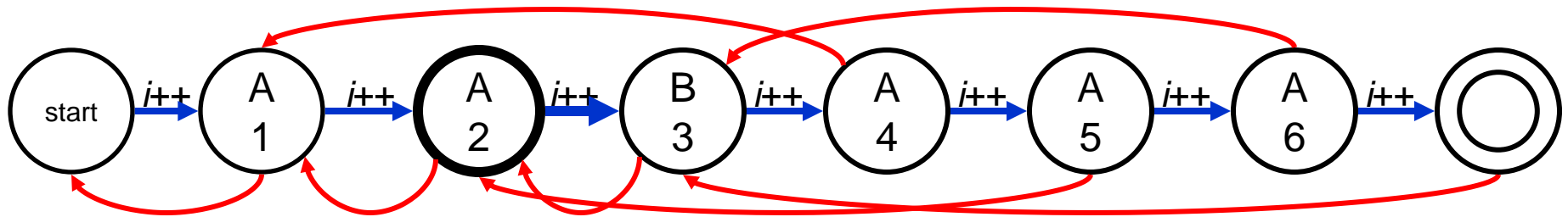
Example: streaming text  $T=aaabaaa$ **a**abaaaa through the automaton

# KMP pattern matching automaton

1	2	3	4	5	6
<b>a</b>	<b>a</b>	<b>b</b>	<b>a</b>	<b>a</b>	<b>a</b>

OF

0	1	0	1	2	2
---	---	---	---	---	---



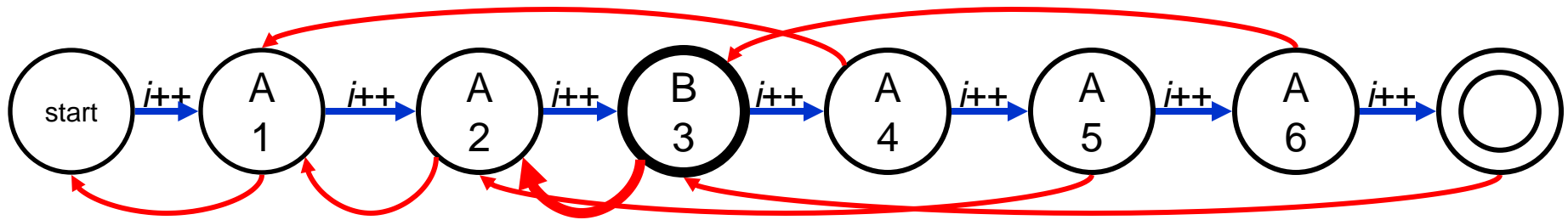
Example: streaming text  $T=aaabaaa$ **a** $abaaaa$  through the automaton

# KMP pattern matching automaton

1	2	3	4	5	6
<b>a</b>	<b>a</b>	<b>b</b>	<b>a</b>	<b>a</b>	<b>a</b>

OF

0	1	0	1	2	2
---	---	---	---	---	---



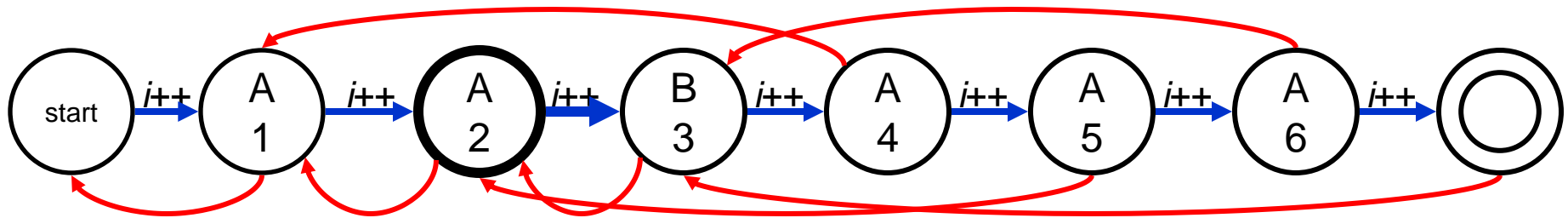
Example: streaming text  $T=aaabaaa**a**baaaa$  through the automaton

# KMP pattern matching automaton

1	2	3	4	5	6
<b>a</b>	<b>a</b>	<b>b</b>	<b>a</b>	<b>a</b>	<b>a</b>

OF

0	1	0	1	2	2
---	---	---	---	---	---



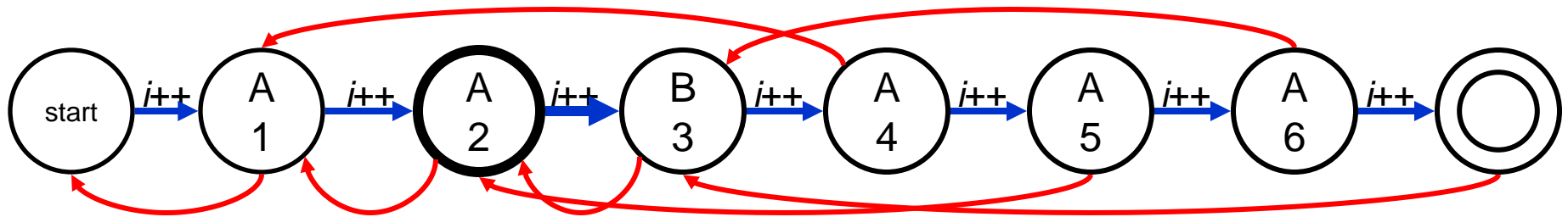
Example: streaming text  $T=aaabaaaa**a**baaaa$  through the automaton

# KMP pattern matching automaton

1	2	3	4	5	6
<b>a</b>	<b>a</b>	<b>b</b>	<b>a</b>	<b>a</b>	<b>a</b>

OF

0	1	0	1	2	2
---	---	---	---	---	---



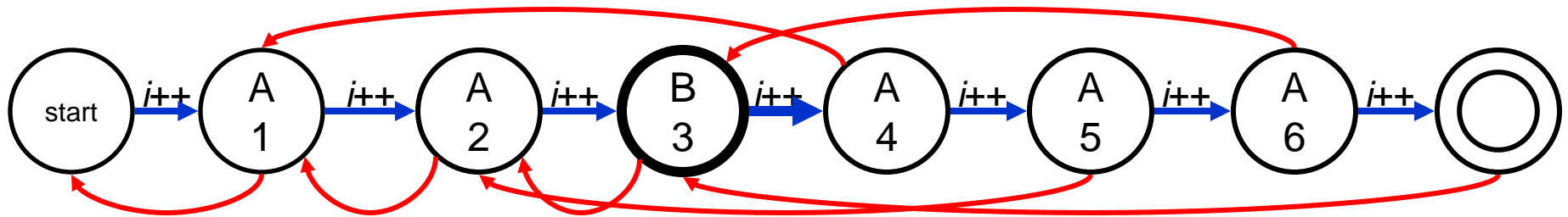
Example: streaming text  $T=aaabaaa**a**baaaa$  through the automaton

# KMP pattern matching automaton

1	2	3	4	5	6
<b>a</b>	<b>a</b>	<b>b</b>	<b>a</b>	<b>a</b>	<b>a</b>

OF

0	1	0	1	2	2
---	---	---	---	---	---



Example: streaming text  $T=aaabaaaa**b**aaaa$  through the automaton

Etc...

---

# Automaton for a set of patterns

- The KMP automaton can be build for a set of patterns
  - In this case we are simultaneously finding the positions of several patterns in  $T$  by streaming  $T$  through the automaton
  - The automaton for a set of patterns is left as an exercise for you and may be chosen as a project (the Aho-Corasick algorithm)
-

---

# References

- [http://en.wikipedia.org/wiki/Knuth-Morris-Pratt\\_algorithm](http://en.wikipedia.org/wiki/Knuth-Morris-Pratt_algorithm)
  - <http://www.ics.uci.edu/~eppstein/161/960227.html>
  - Dan Gusfield. Algorithms on strings, trees, and sequences. Computer science and computational biology. Cambridge University press, 1999.
-