Exhaustive substring search. Algorithm by Knuth, Morris, Pratt (KMP)

Lecture 04.02

by Marina Barsky

Strings

STRINGS ARE NATURAL GROUPINGS OF SYMBOLS INTO SEQUENCES, WHERE THE ORDER HAS A SPECIAL SIGNIFICANCE

bad salad

$\neq$

sad ballad

a b d l s

Strings encode life

"*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

Useful definitions: string and substring

- 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*

Useful definitions: prefix and suffix

- *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*

b	**a**	**n**	**a**	**n**	**a**
1	*2*	*3*	*4*	*5*	*6*

What is Suffix 4?

What is Suffix 1?

Useful definitions: prefix and suffix

- *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 N

b	**a**	**n**	**a**	**n**	**a**
1	*2*	*3*	*4*	*5*	*6*

What is Prefix 4?

What is Prefix 1?

What is Prefix 0?

Useful definitions: proper substrings

- *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 N

- *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

Useful definitions: proper substrings

- *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 N

- A *proper* substring, prefix, suffix of S is respectively a substring, prefix, suffix that is neither the entire string S nor the empty string

b	**a**	**n**	**a**	**n**	**a**
1	*2*	*3*	*4*	*5*	*6*

Is Prefix 1 a proper prefix?

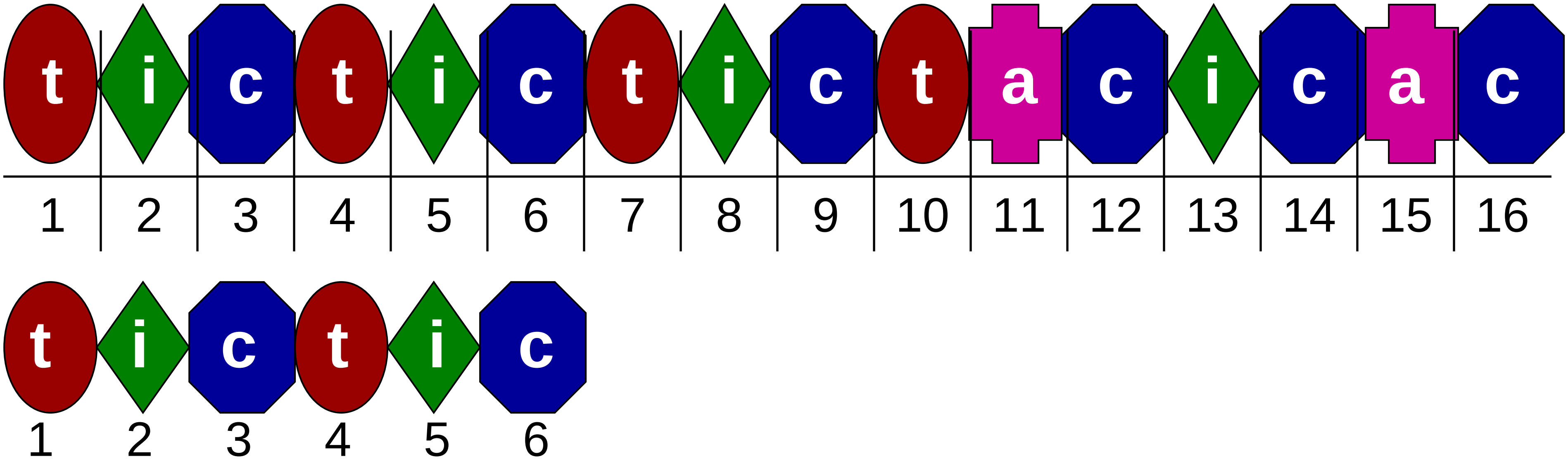
Is Prefix 0 a proper prefix?

Is Suffix 1 a proper suffix?

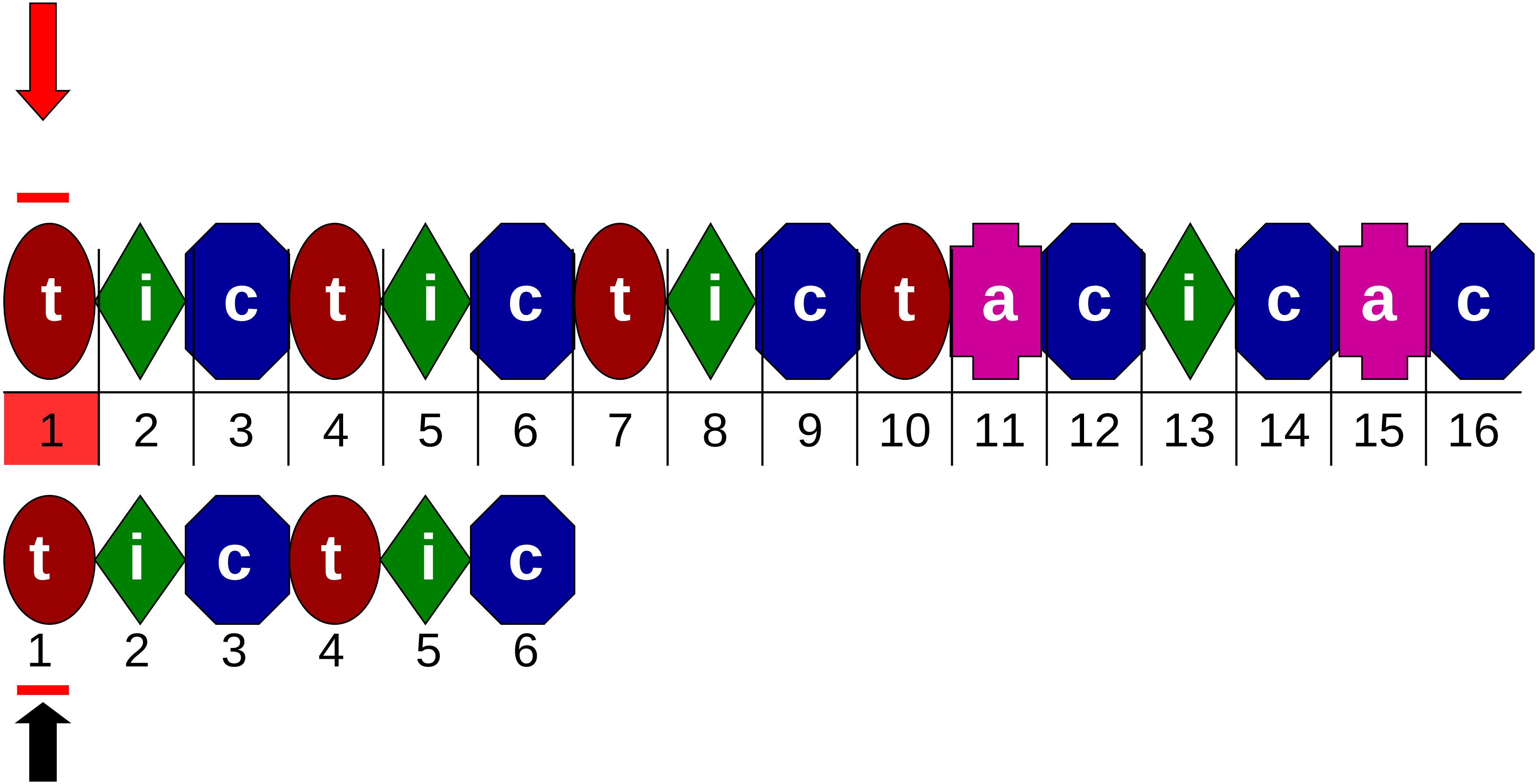
Pattern matching problem

- 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*

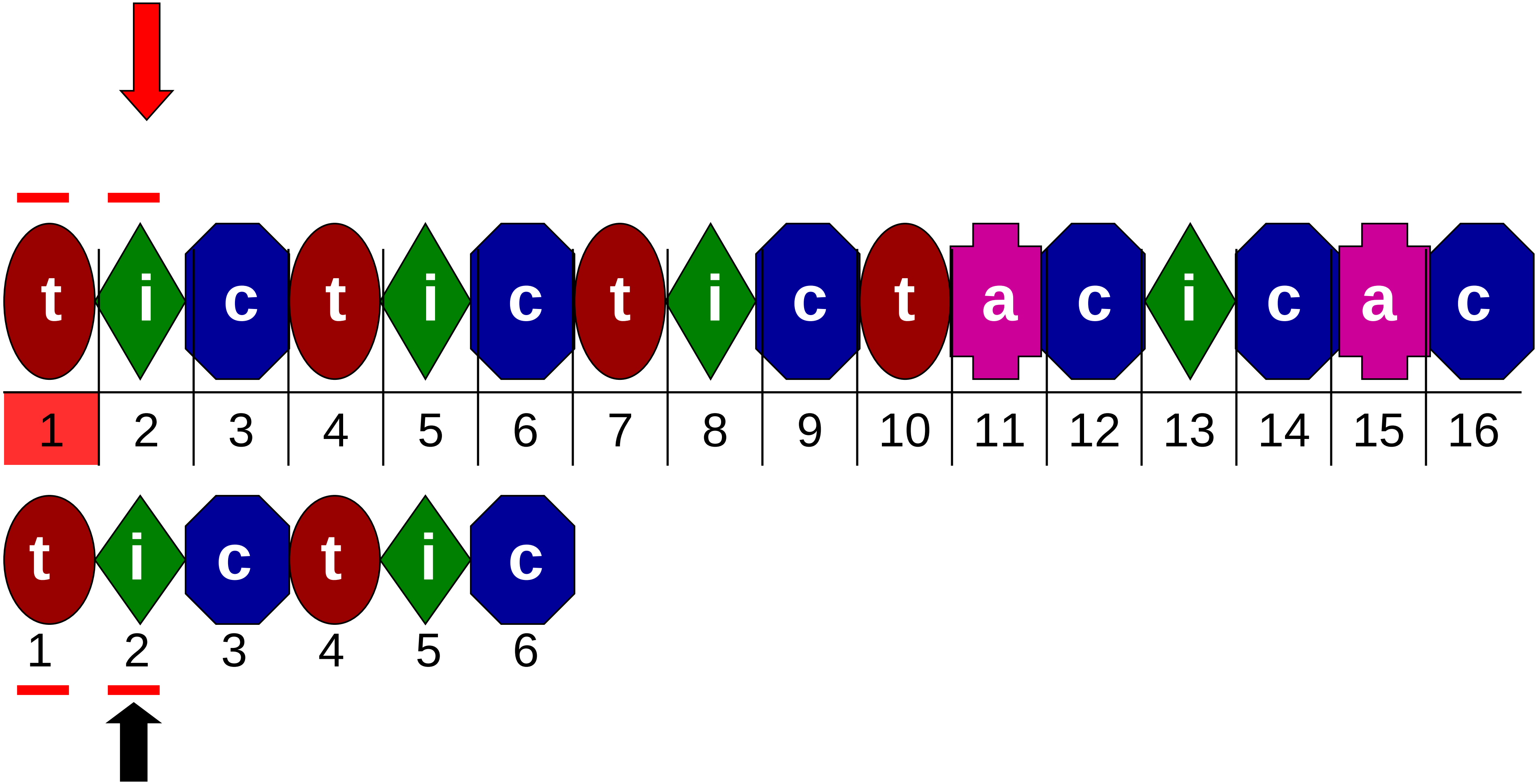
Naïve exhaustive search



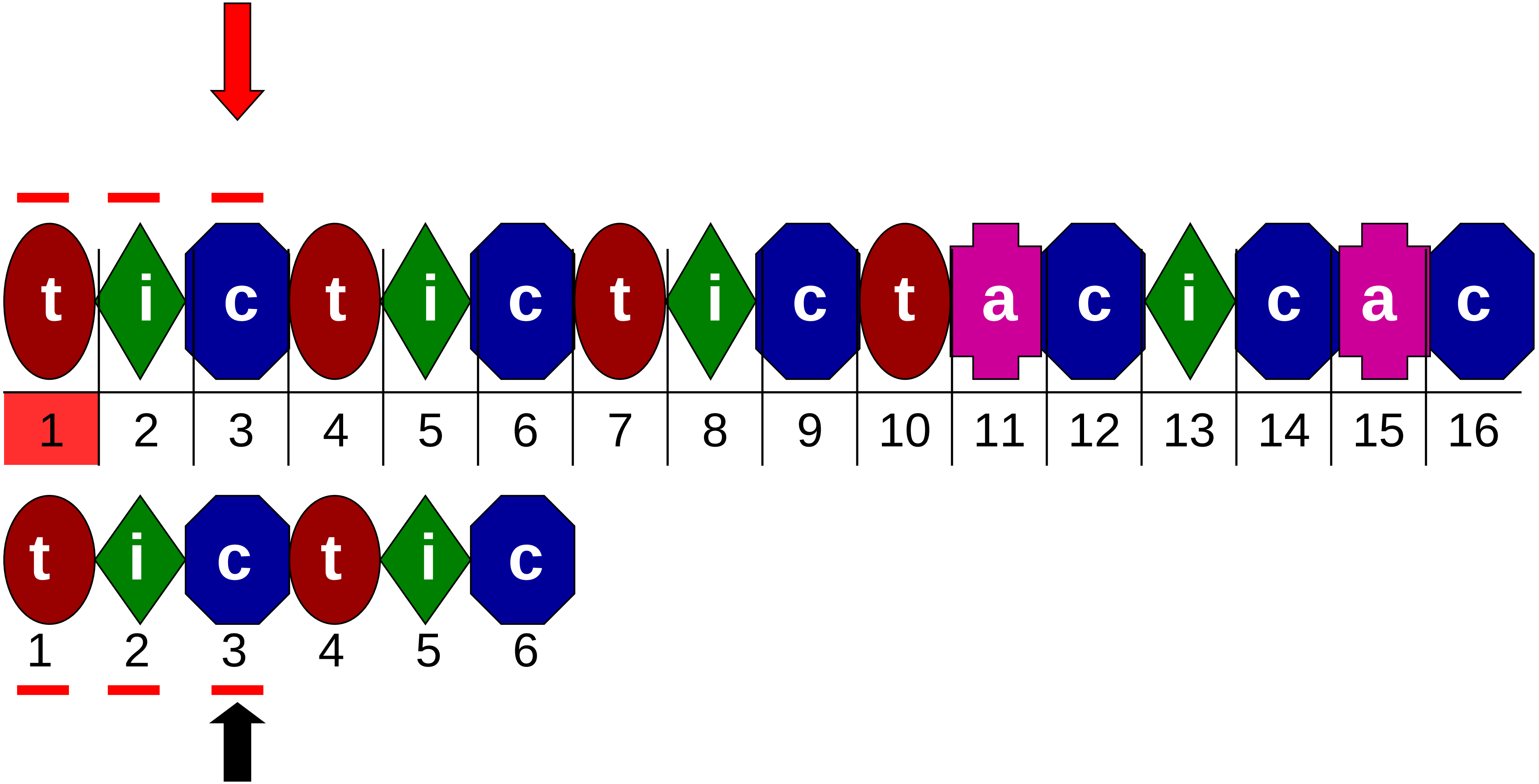
Naïve exhaustive search



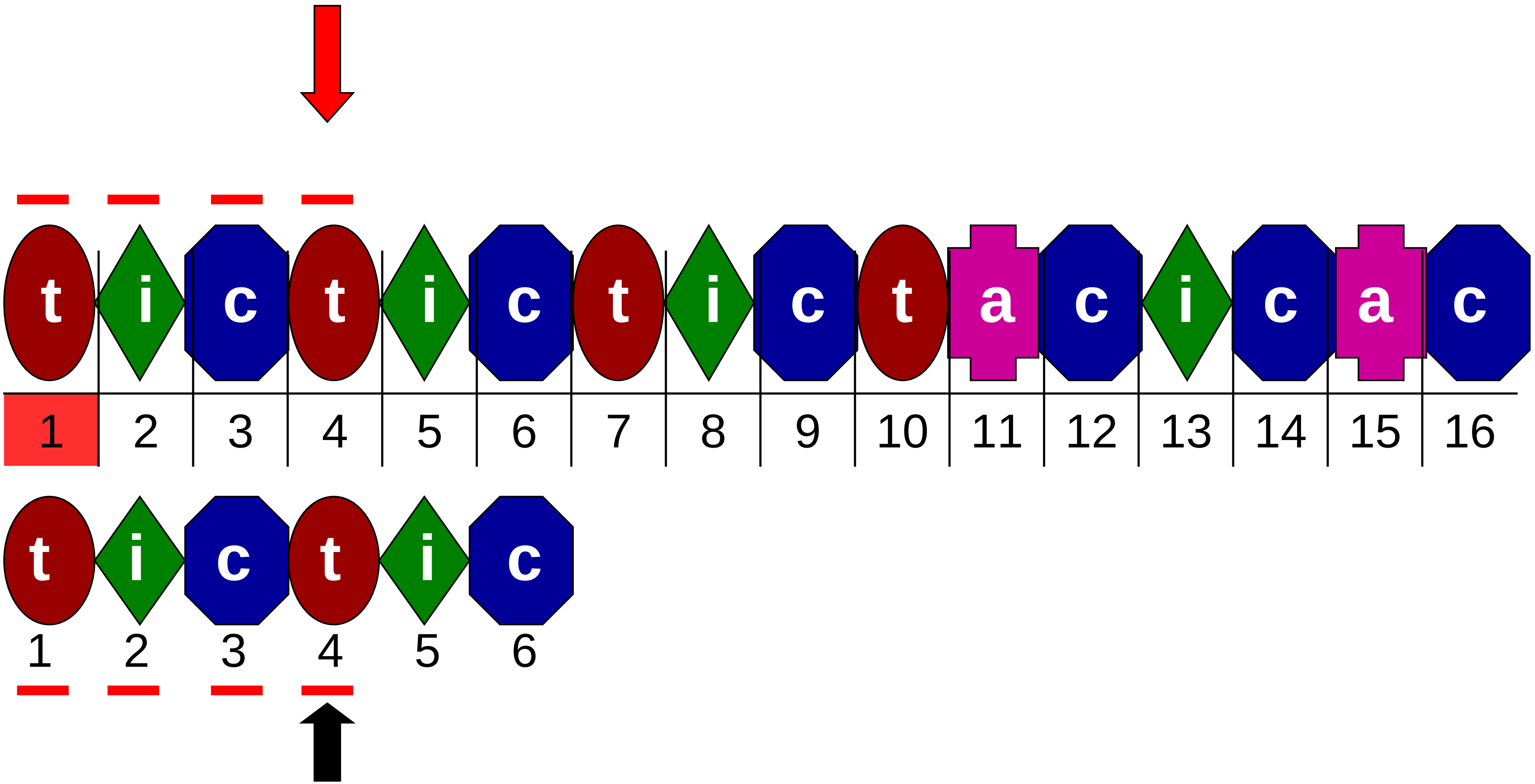
Naïve exhaustive search



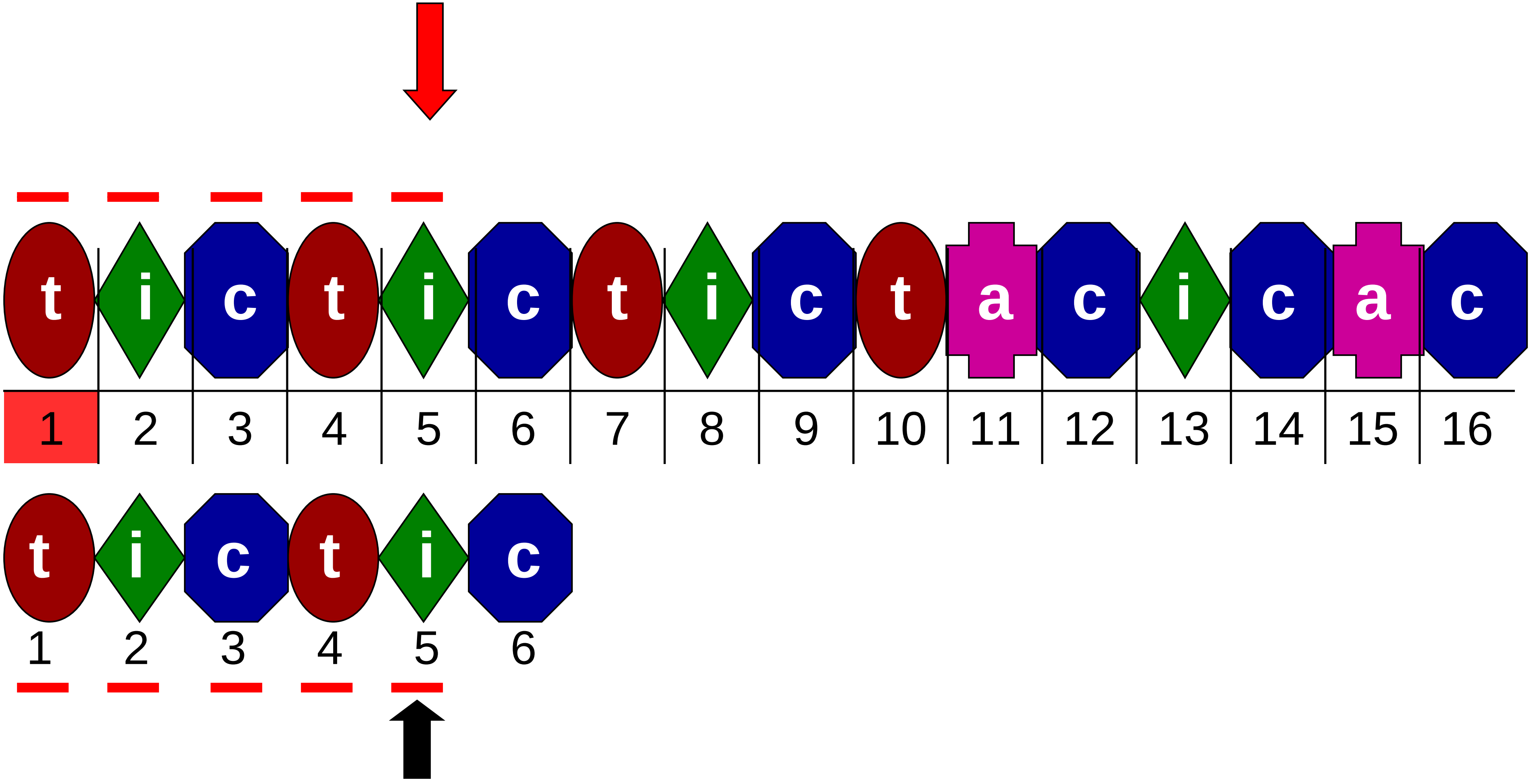
Naïve exhaustive search



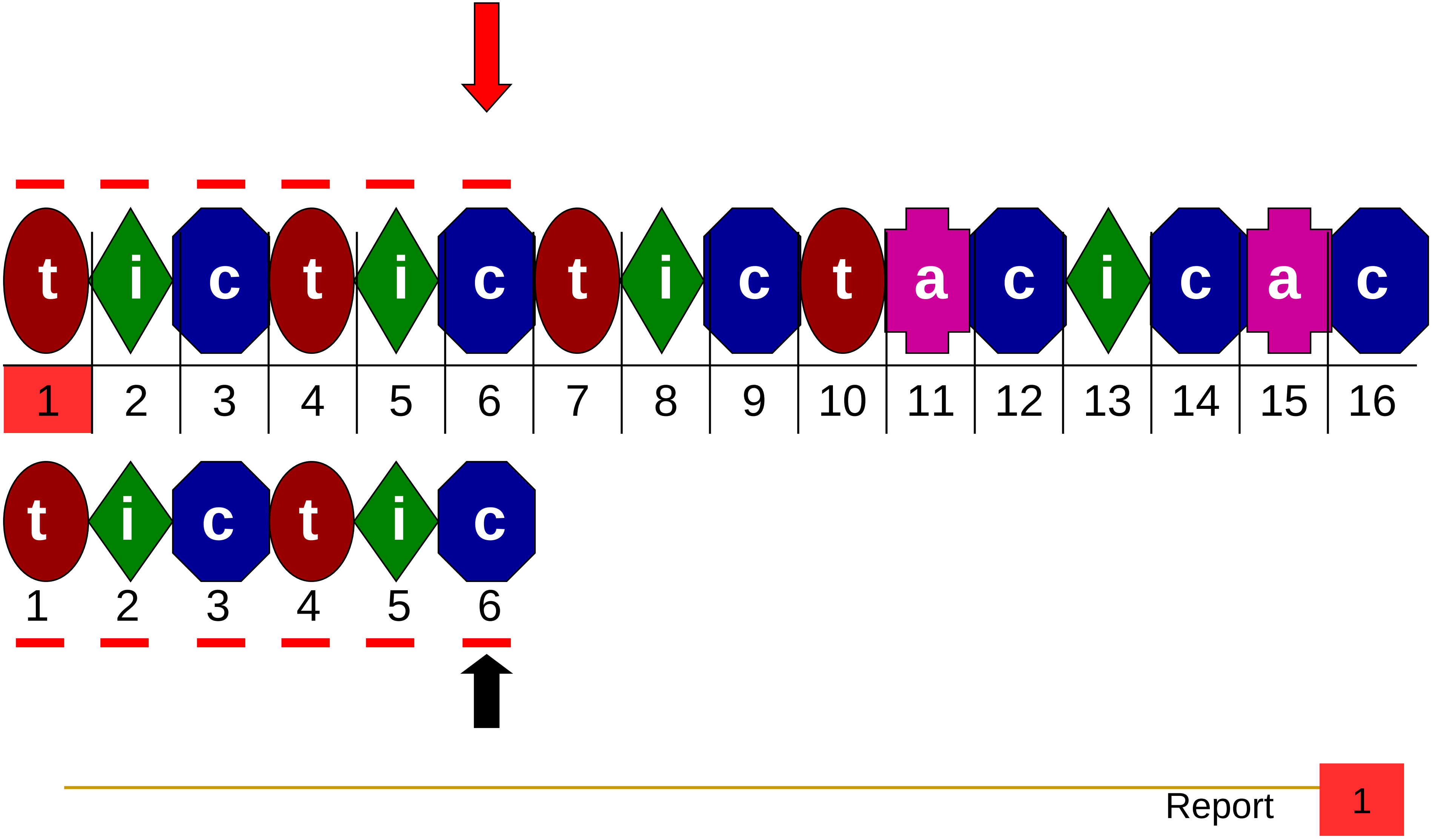
Naïve exhaustive search



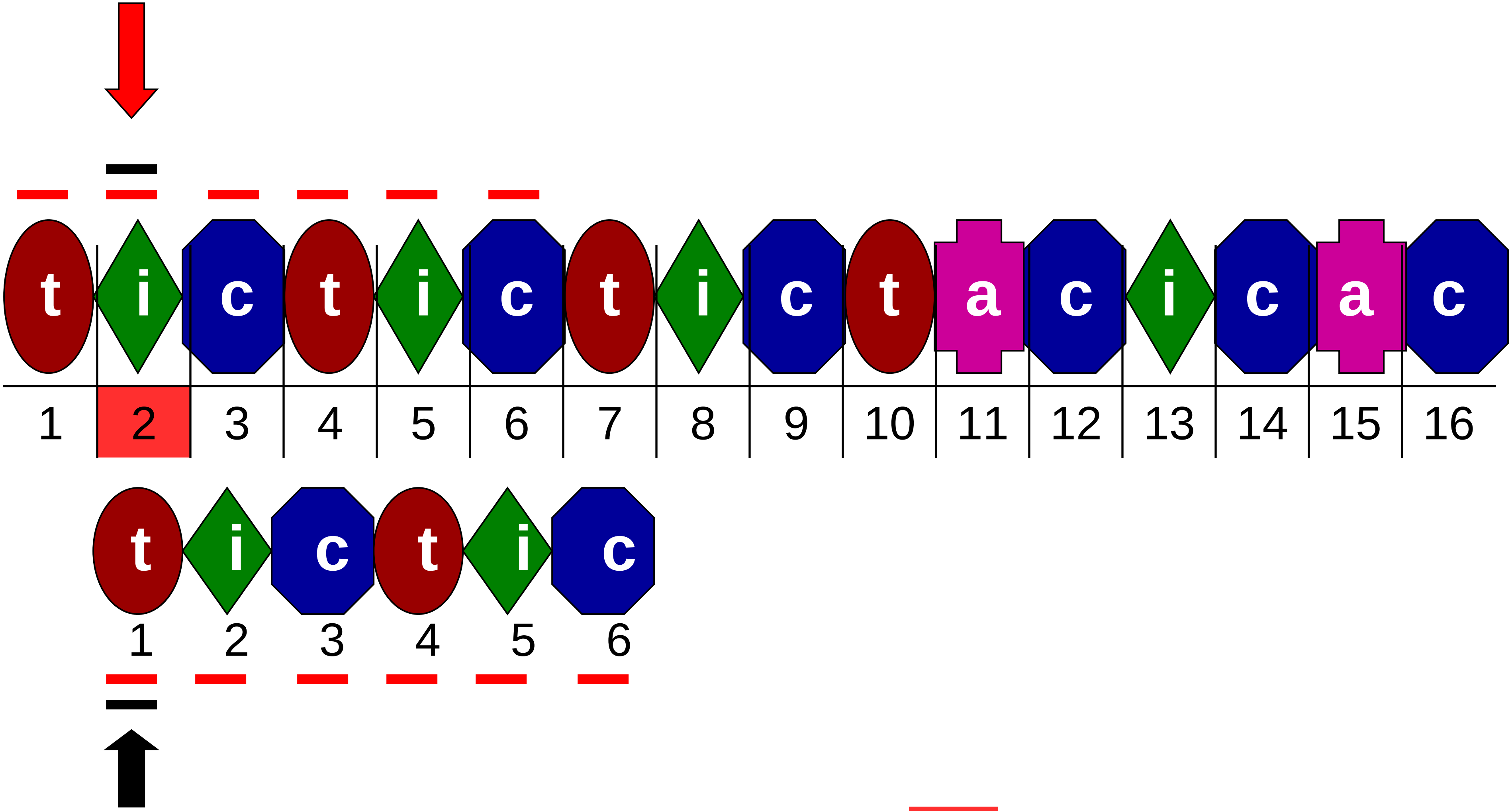
Naïve exhaustive search



Naïve exhaustive search

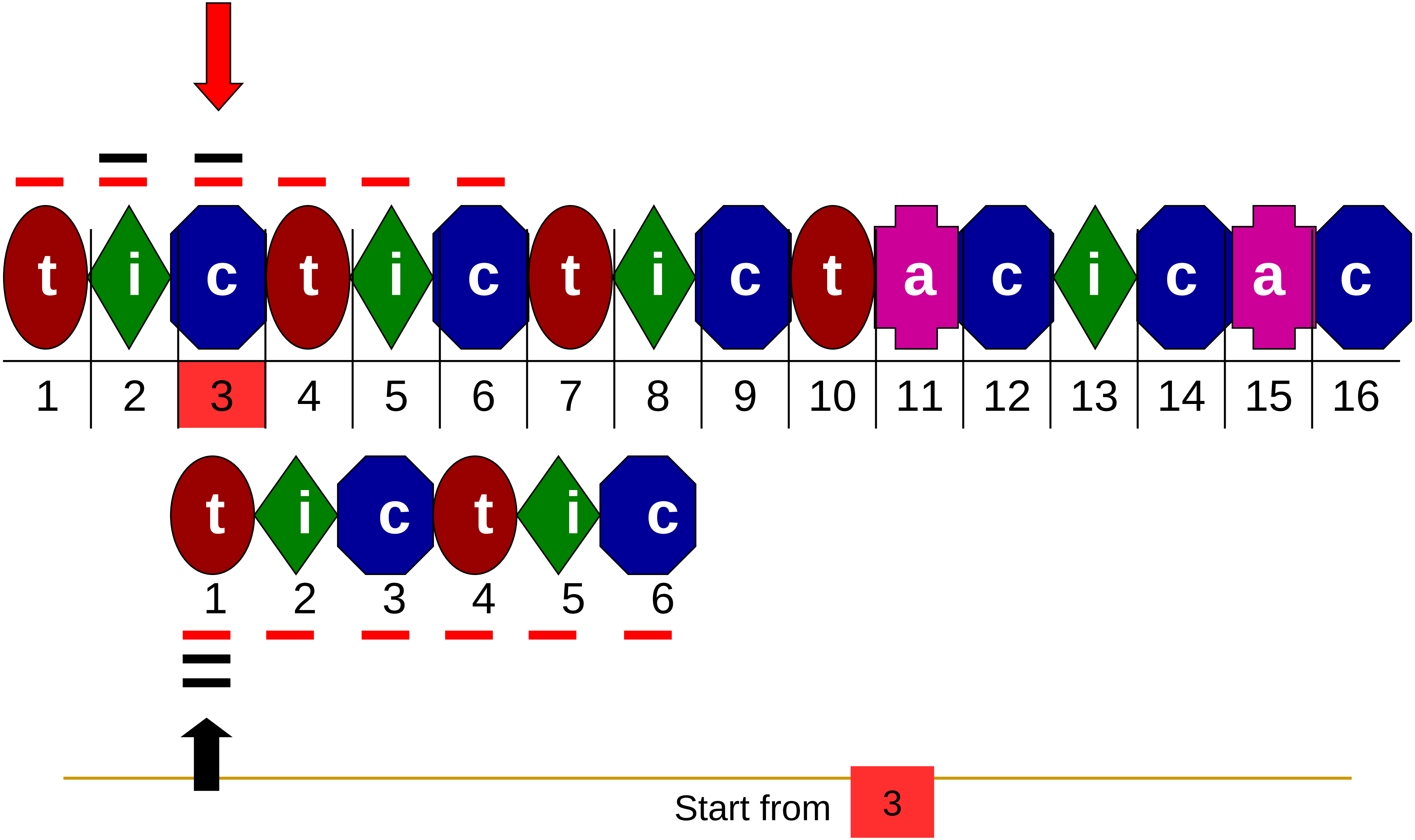


Naïve method – what next?

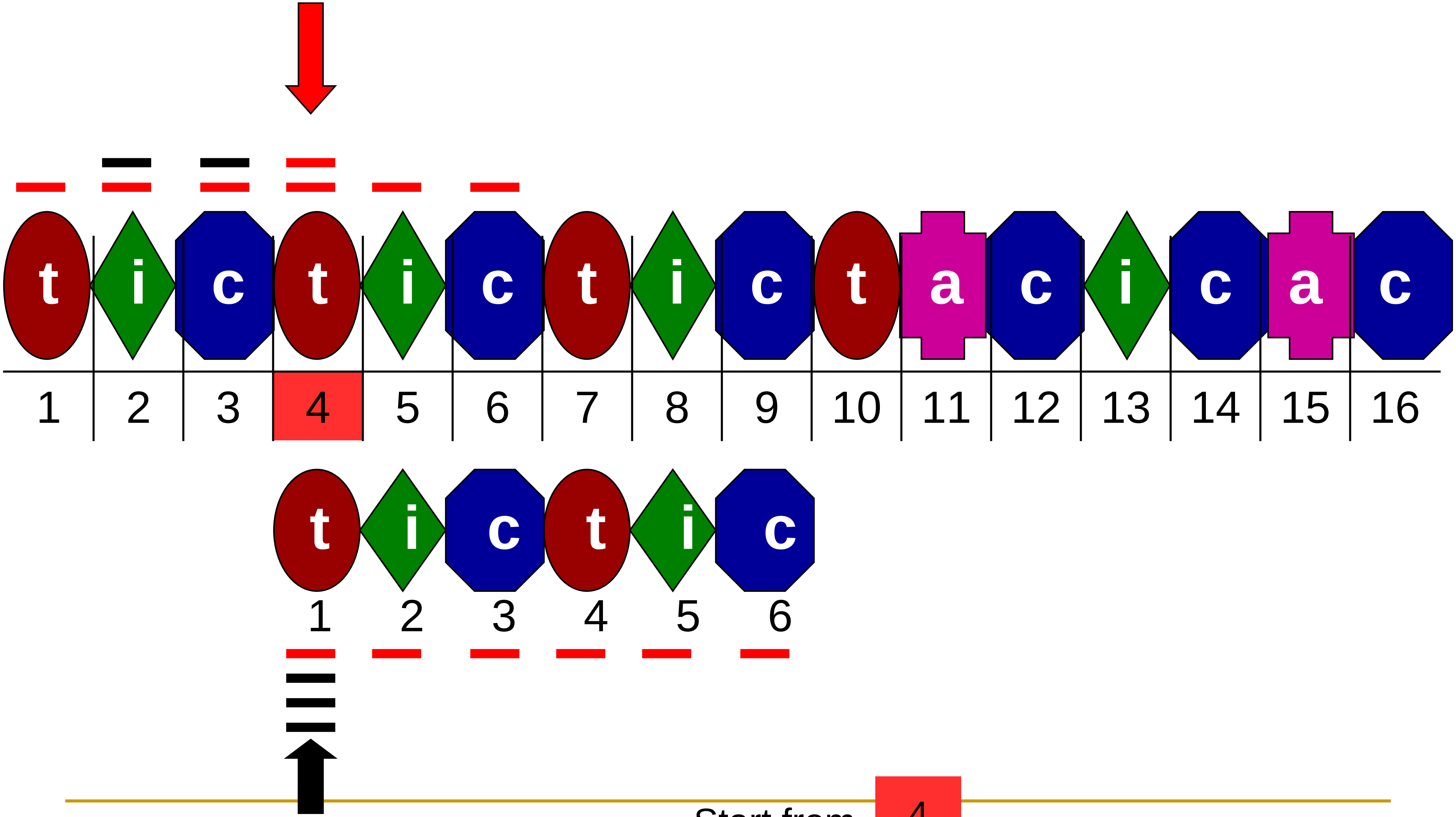


Start from 2

Naïve method – what next?

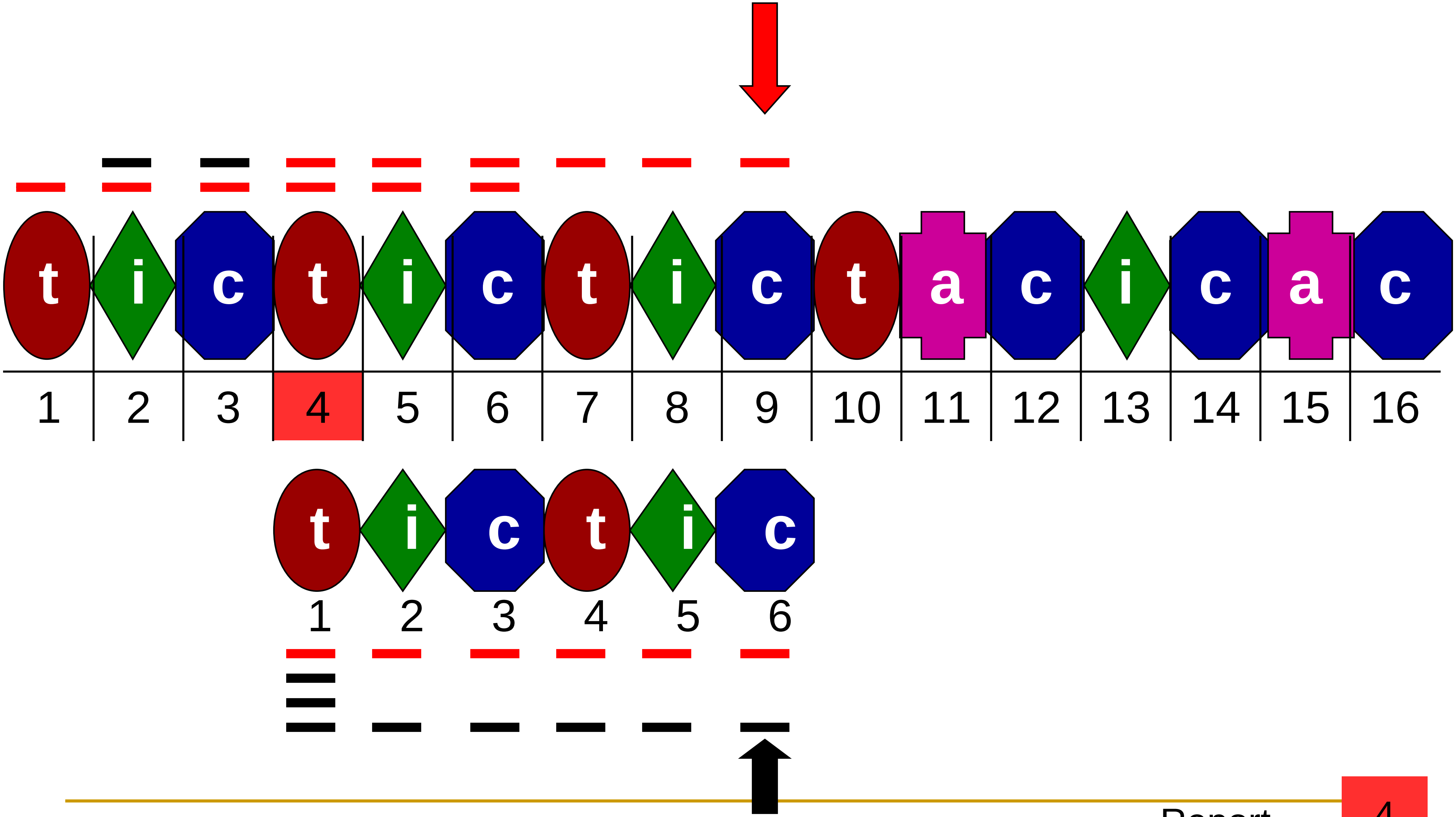


Naïve method – what next?



Start from 4

Naïve method – continue…



Report

Naïve method – time complexity

- How many character comparisons in total?
- How did you compute the value?
- Compute how many comparisons are required for T=*aaaaaaaaaa* (N=10) and P=*aaa* (M=3)

→ In the worst case, we start from each position i of T (there are N such positions), and, for each i, compare M characters

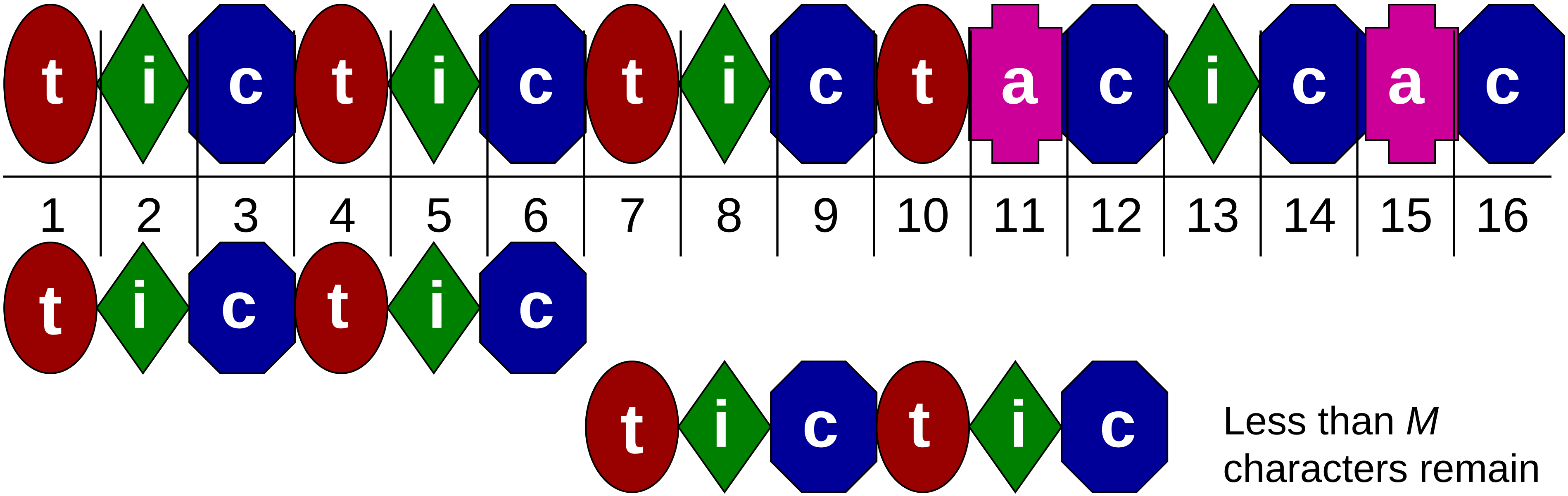
→ For T=*aaaaaaaaaa* (N=10) and P=*aaa* (M=3) there are exactly 24 comparisons, $M*(N-M+1)$

→ The time complexity of the naïve algorithm is $O(MN)$

Can we do better? Motivation

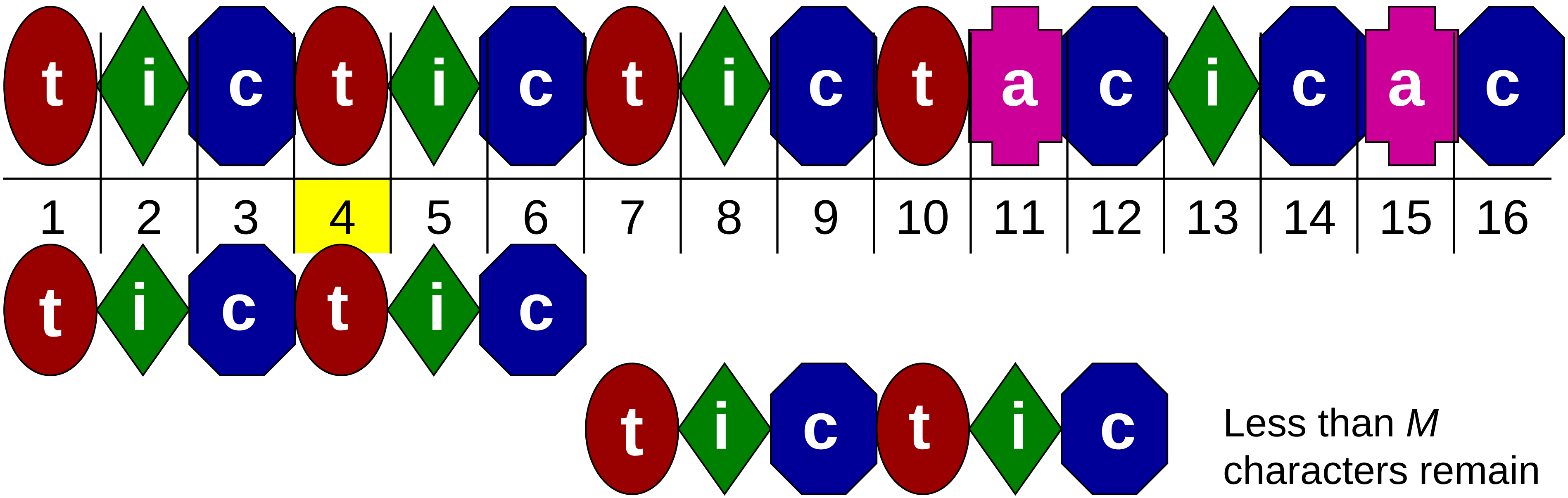
- A standard fetching time from sequential RAM is 358 MB values per second (ref).
- If we have 10 random sets of sequenced fragments from 3GB human genome, then we need to search the text of a total size $3*10^{10}$, which can be sequentially accessed in approximately $3*10^{8}$ values per second. We 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 100!
- We want the pattern search algorithm to perform at least in time O(N).

Dream goal: each character of T is examined at most once



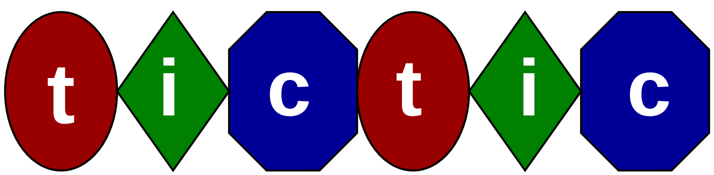
Is this algorithm correct?

Incorrect algorithm



Less than *M* characters remain

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

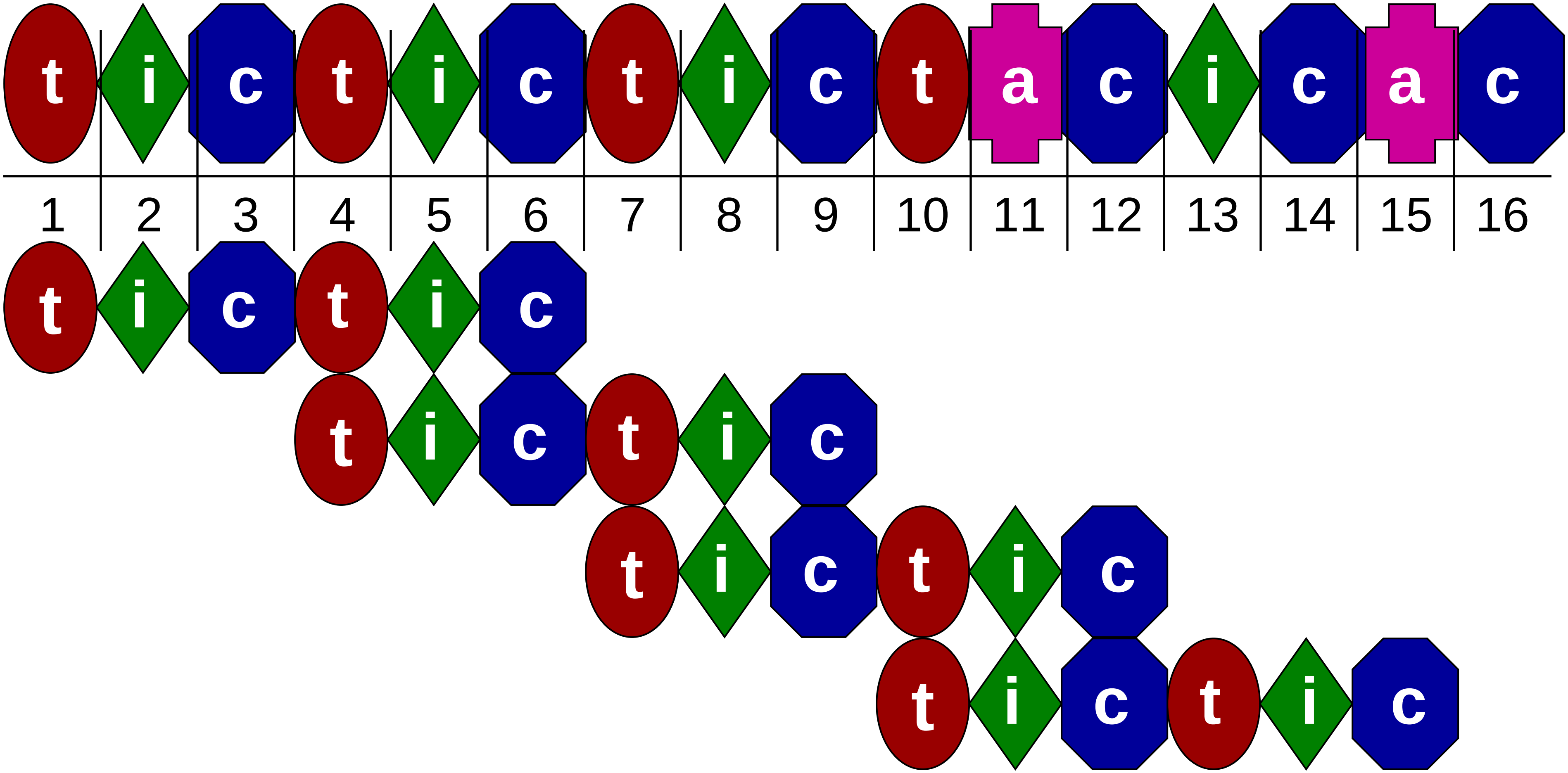


Shifting heuristics

- If we failed to align the next character $P[j]$ of P with the current character of T, start the next comparison from the next occurrence of a character $P[1]$ to the left from j

- How do we know the position in T of such a character?

Shifting heuristics



Seems good!

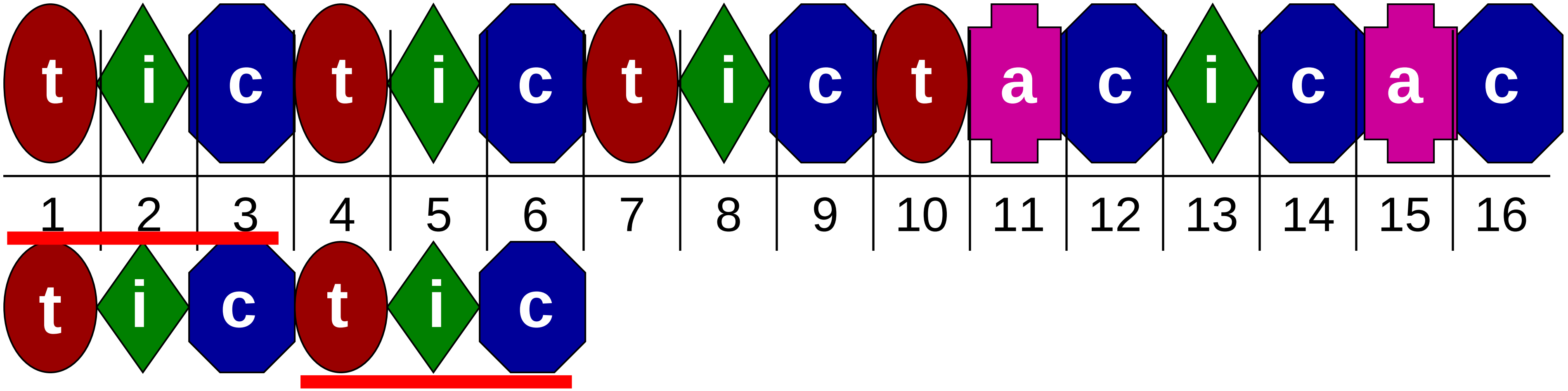
Shifting heuristics

- What about our worst-case example: T=aaaaaaaaaa (N=10) and P=aaa (M=3)?

KMP idea

- When we have aligned the prefix of *P* with *k* characters of *T*, we know what these first *k* characters of *T* are (they are equal to those of the prefix *P*[1…*k*] of *P*).
- From this information we can deduce the place where to start the next comparison.

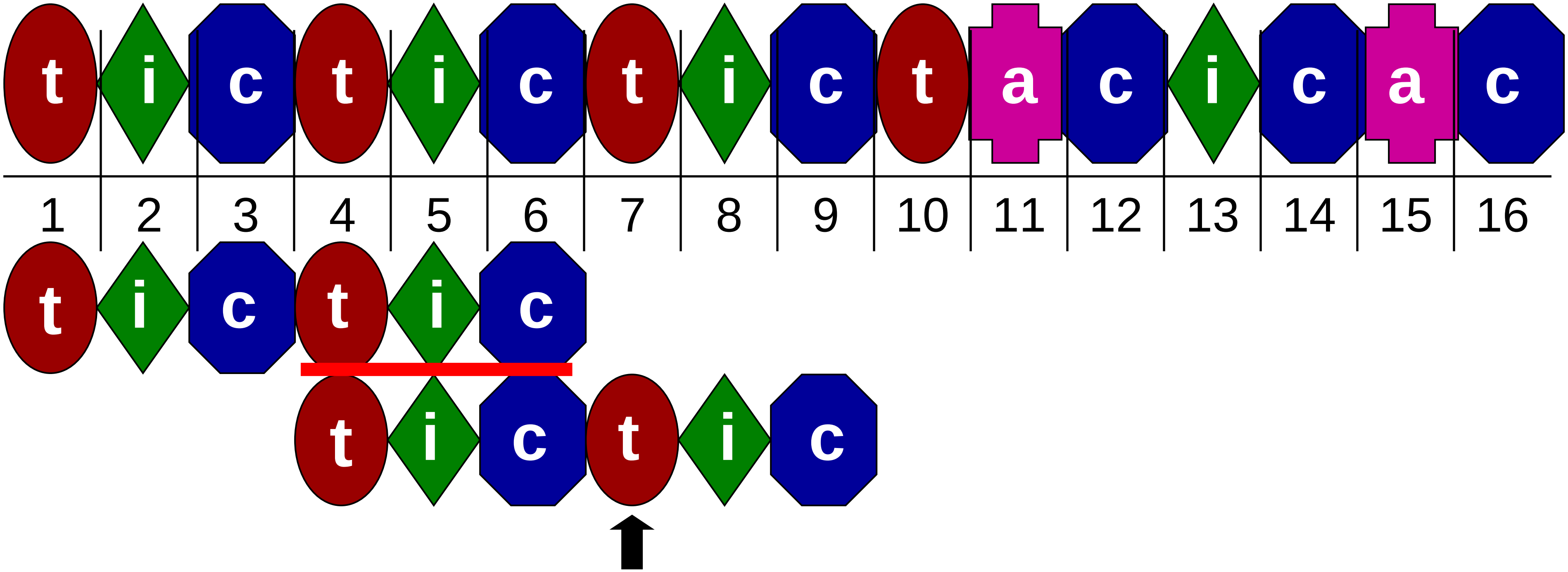
KMP intuition



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

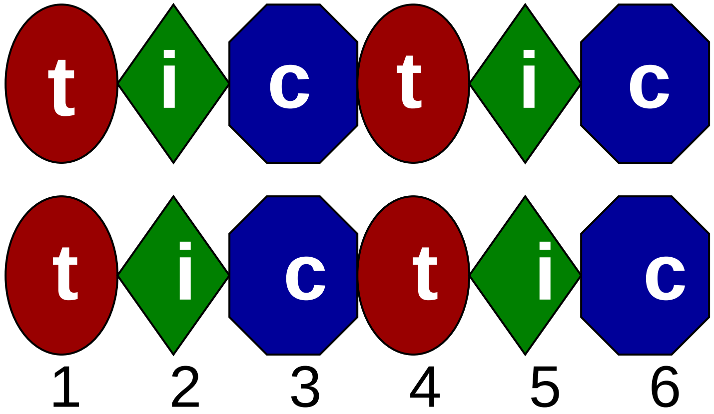


Therefore we can set a start of the next comparison to 3 positions backwards from the current position, and we **don’t need to compare the first 3 characters again**, since we know that they match

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

In this case, we never go back to look at characters of *T* that were already compared.

KMP intuition – overlap function for P

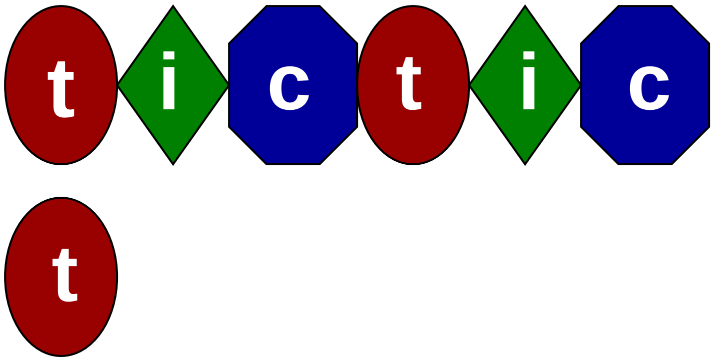


In order to know where to position the start of the next comparison, we need to know the values of an *overlap function* for *P*, namely:

For each position *j* in *P*, the maximal length of a substring which is at the same time a **proper** prefix of *P* and a **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.*

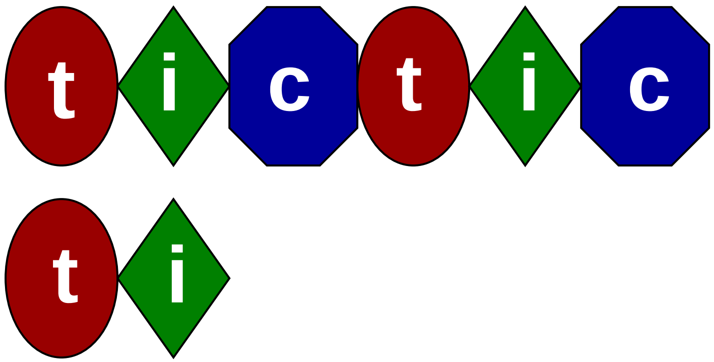
KMP intuition – overlap function for P



1	2	3	4	5	6
0					

For j=1, OF=0 (*t* is not a ***proper*** suffix of a substring *t*, it is the entire *t!*)

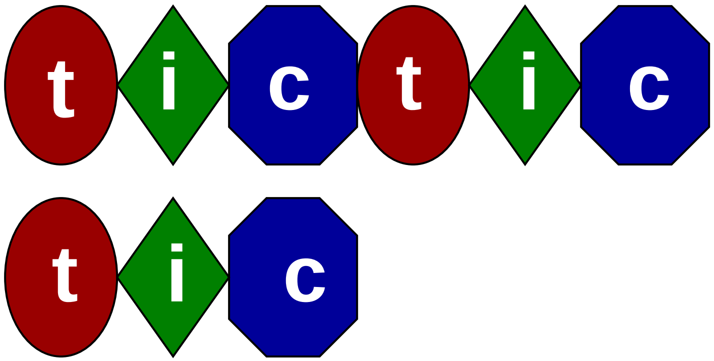
KMP intuition – overlap function for P



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 overlap with any prefix of *ti*)

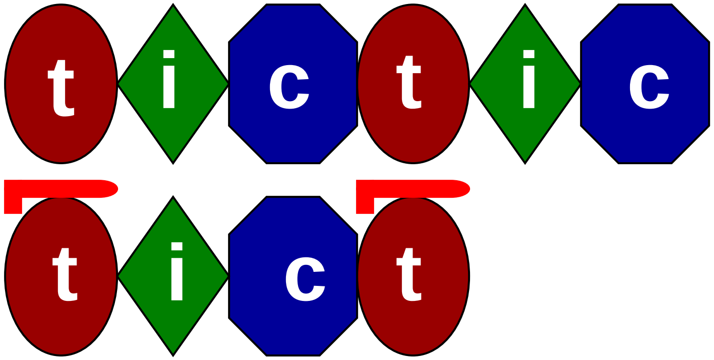
KMP intuition – overlap function for P



1	2	3	4	5	6
0	0	0			

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

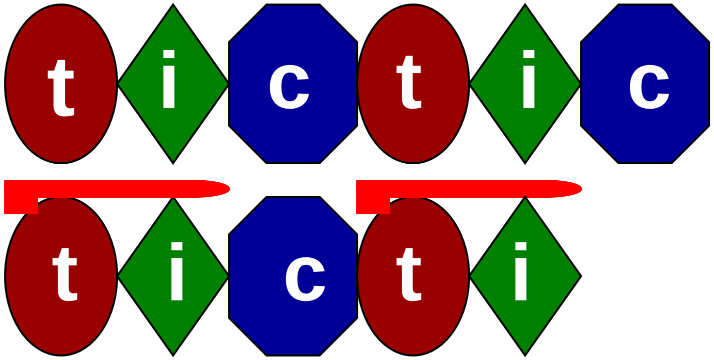
KMP intuition – overlap function for P



1	2	3	4	5	6
0	0	0	1		

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

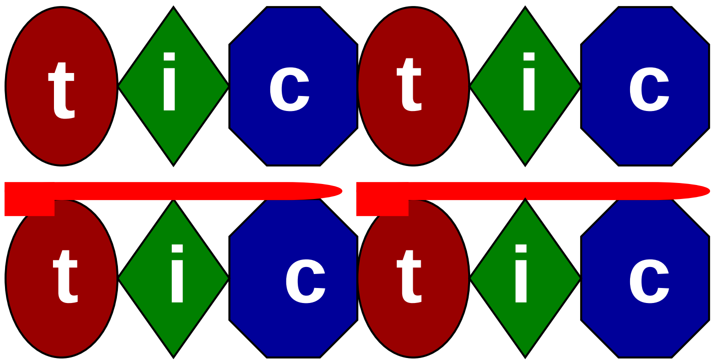
KMP intuition – overlap function for P



1	2	3	4	5	6
0	0	0	1	2	

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

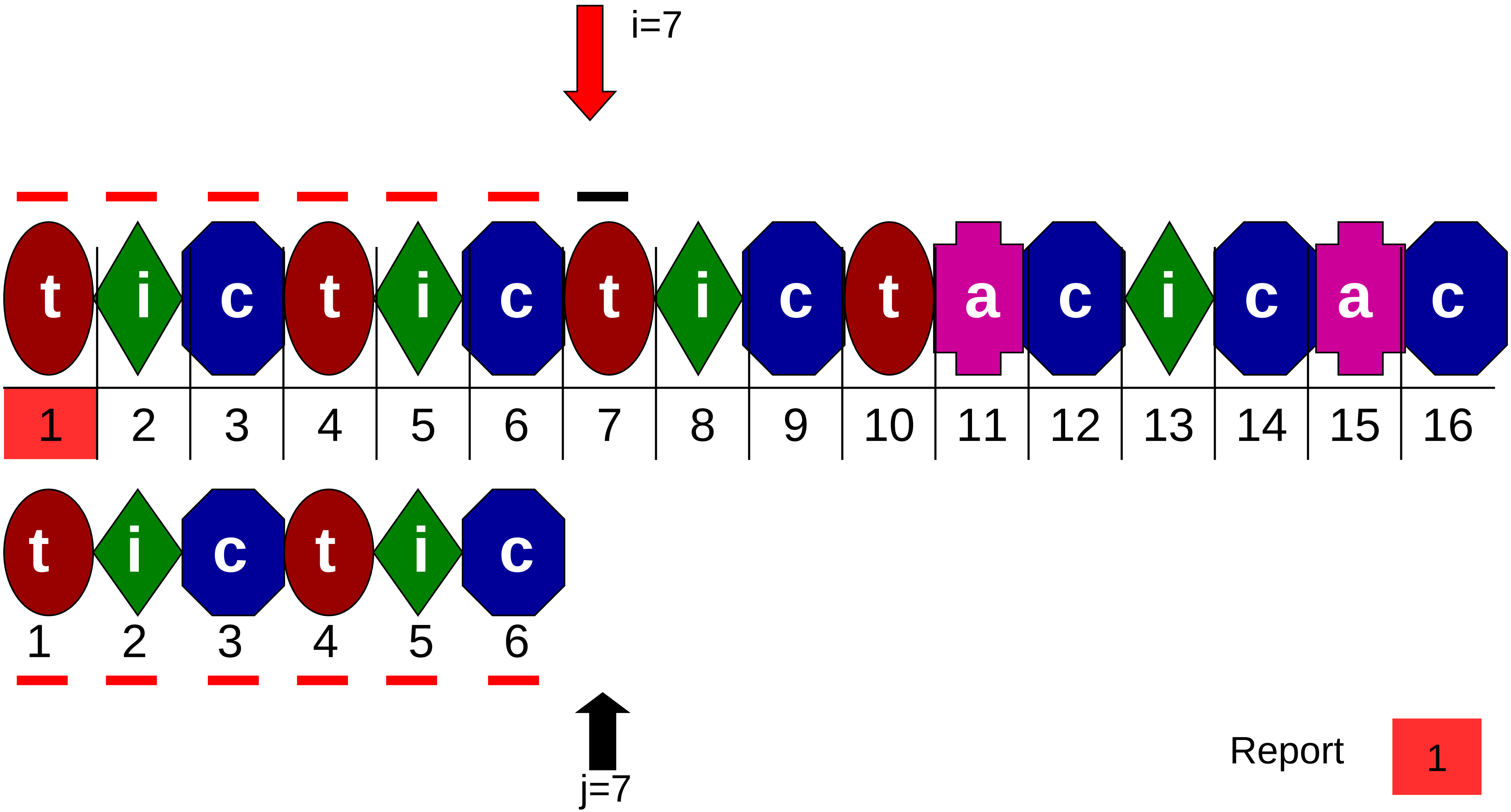


1	2	3	4	5	6
0	0	0	1	2	3

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 pre-computed

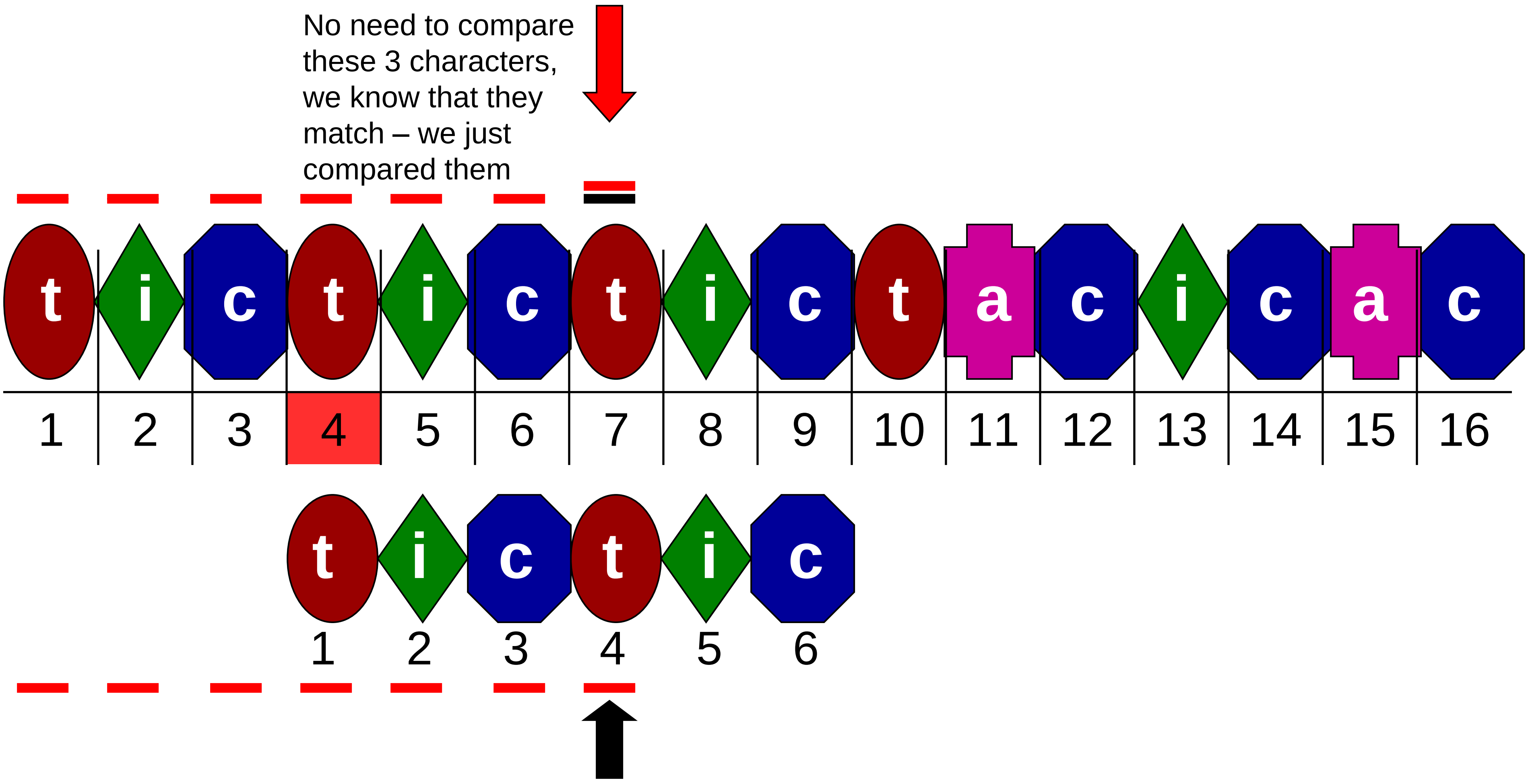
KMP search: match found



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)

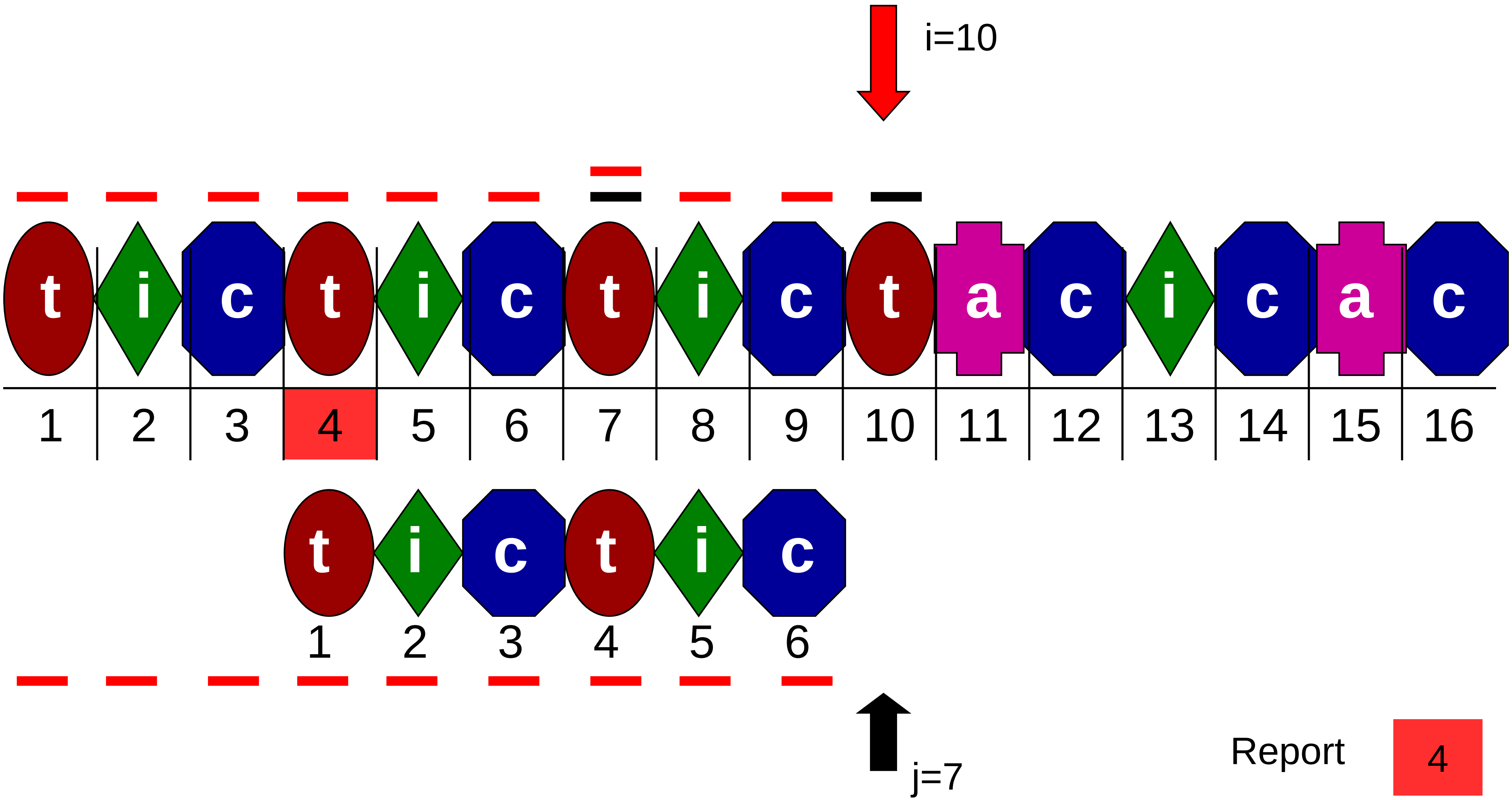
KMP search: overlap 3



0	0	0	1	2	3

Next alignment starts at: k=4

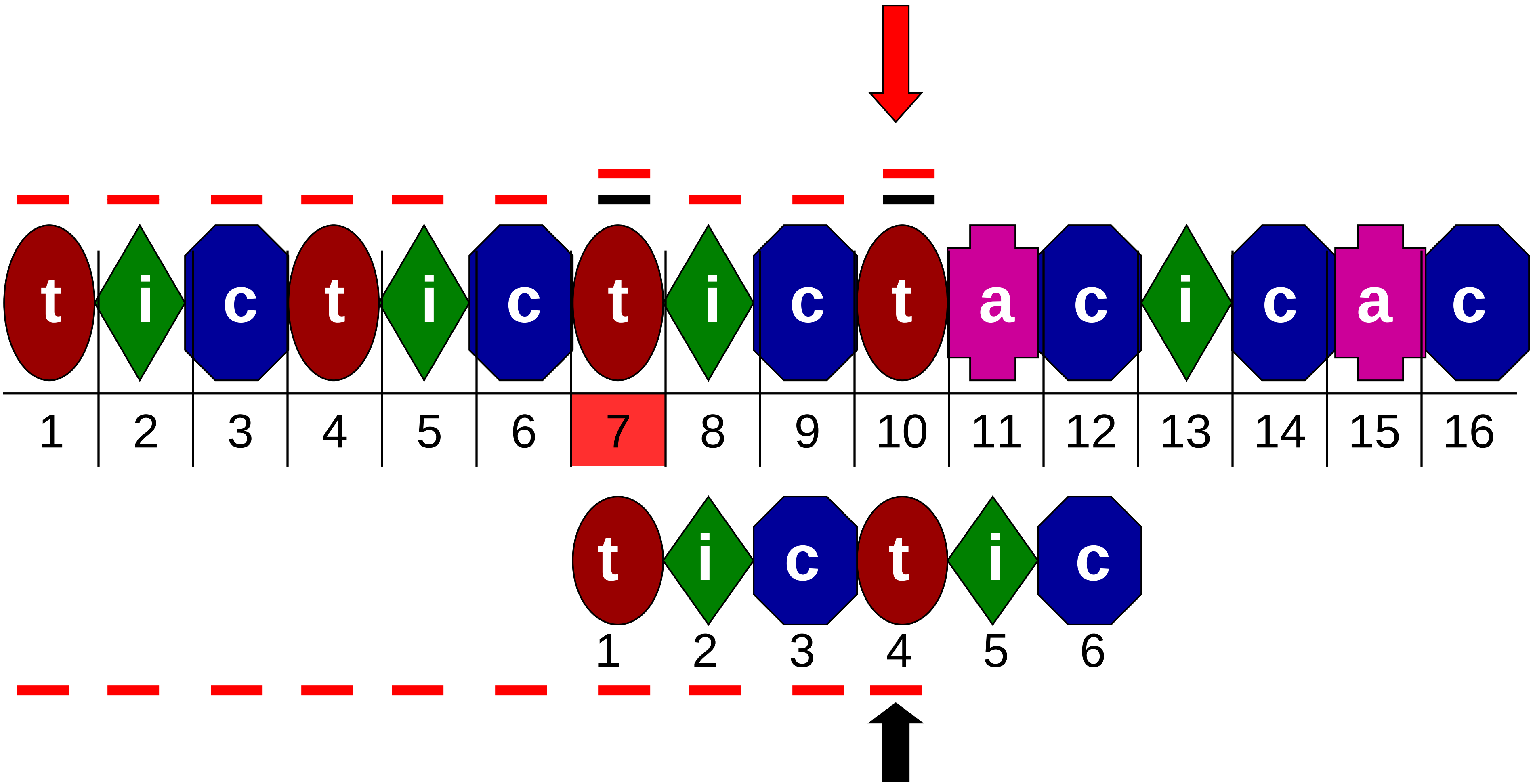
KMP search: overlap 3



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) = 10-3=7

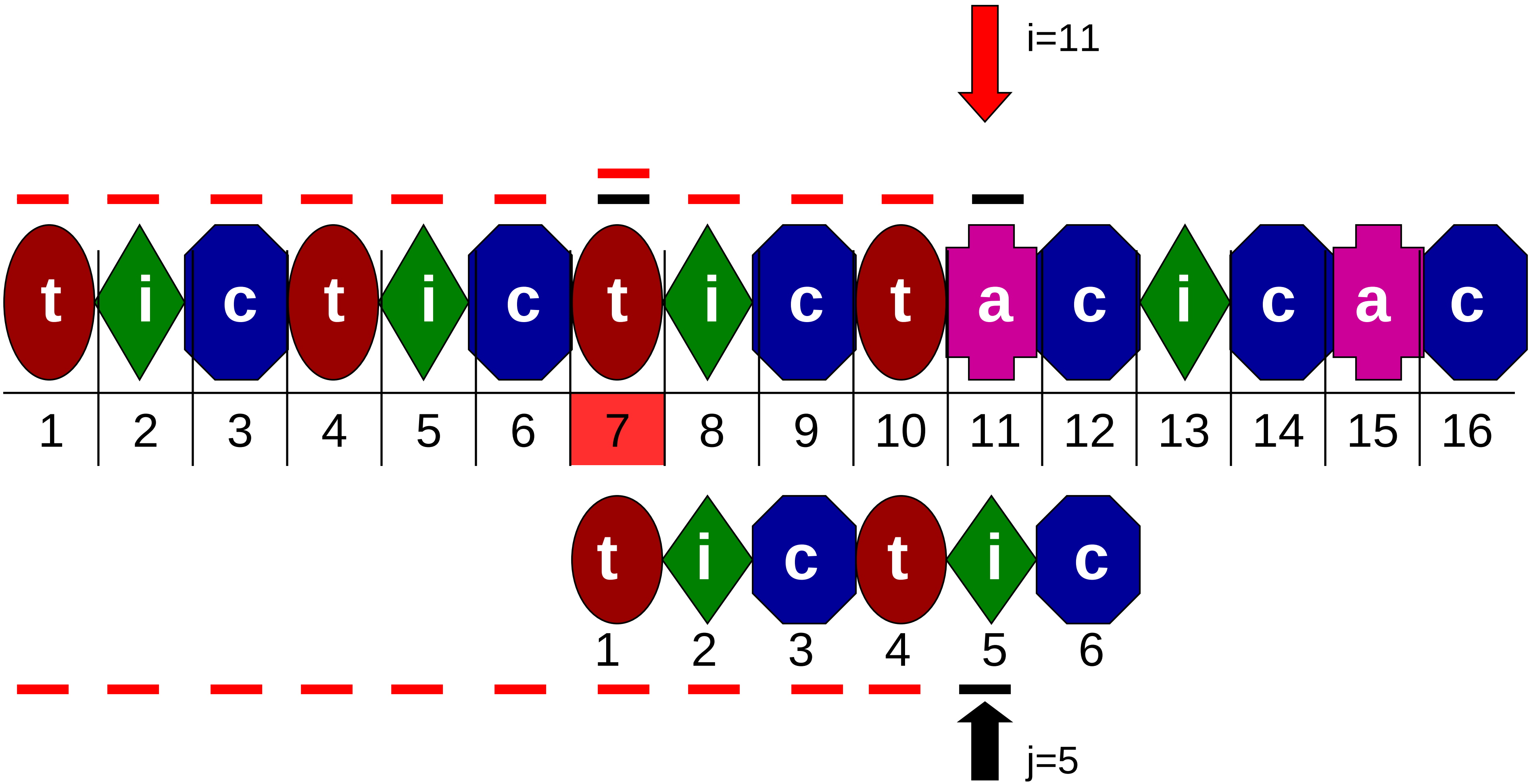
KMP search



0	0	0	1	2	3

Continue comparing T[10] and P[4]

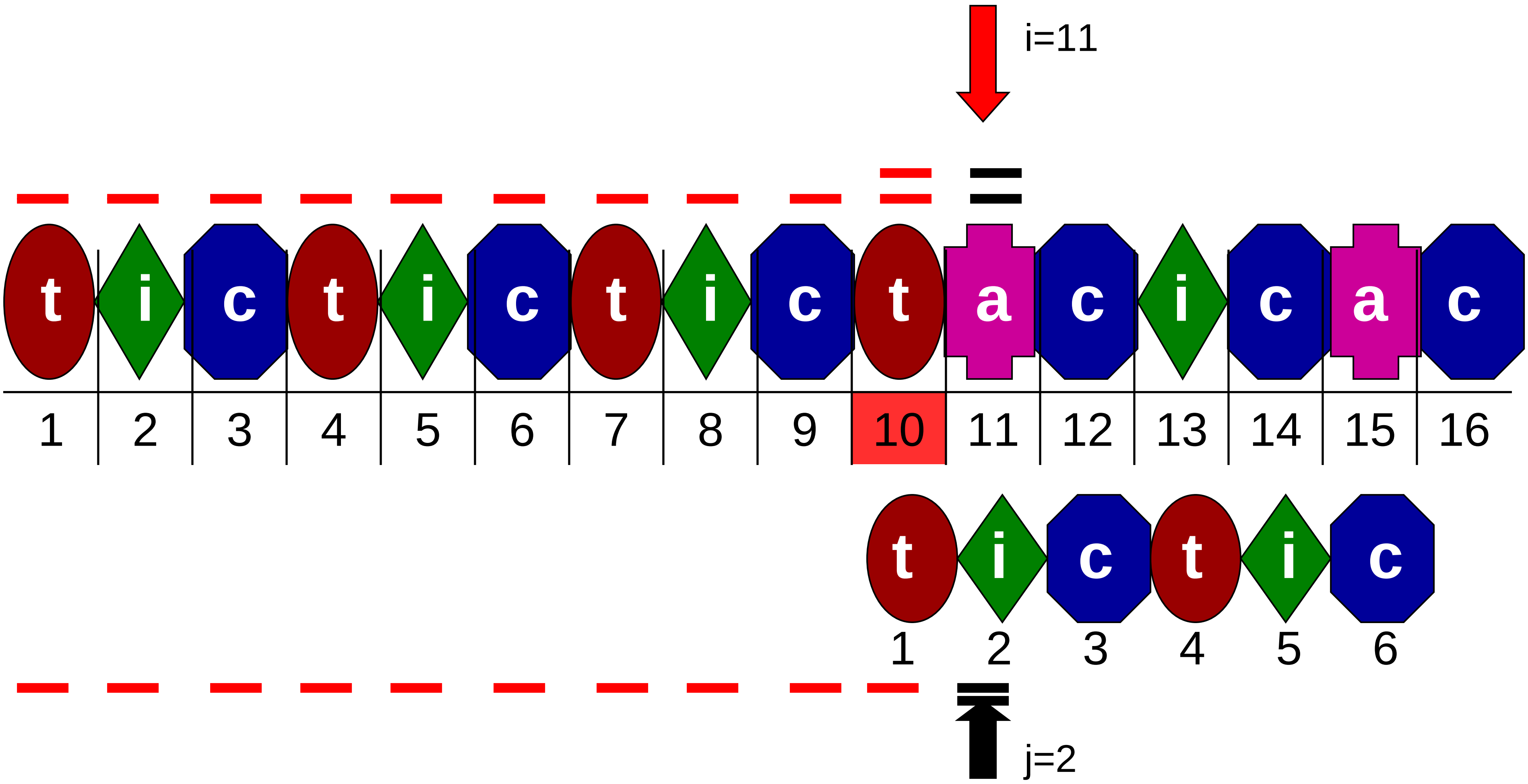
KMP search: overlap 1



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-1)=10, and the first character is already matched.

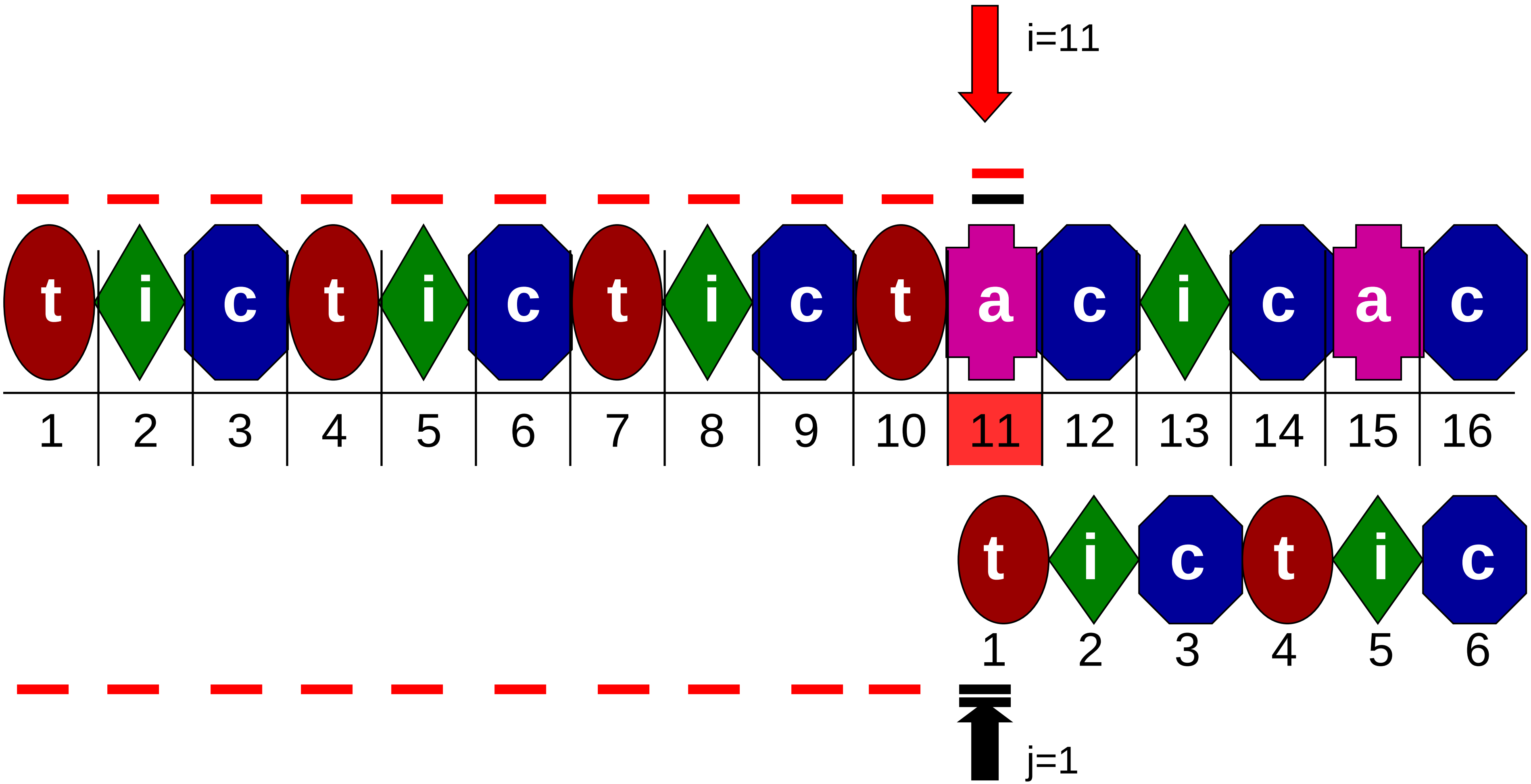
KMP search: overlap 0



0	0	0	1	2	3

Here we only matched with the first character of P, the value OF(1)=0, thus we don't use any info to shift i. We reset pattern position j to 1, without changing i.

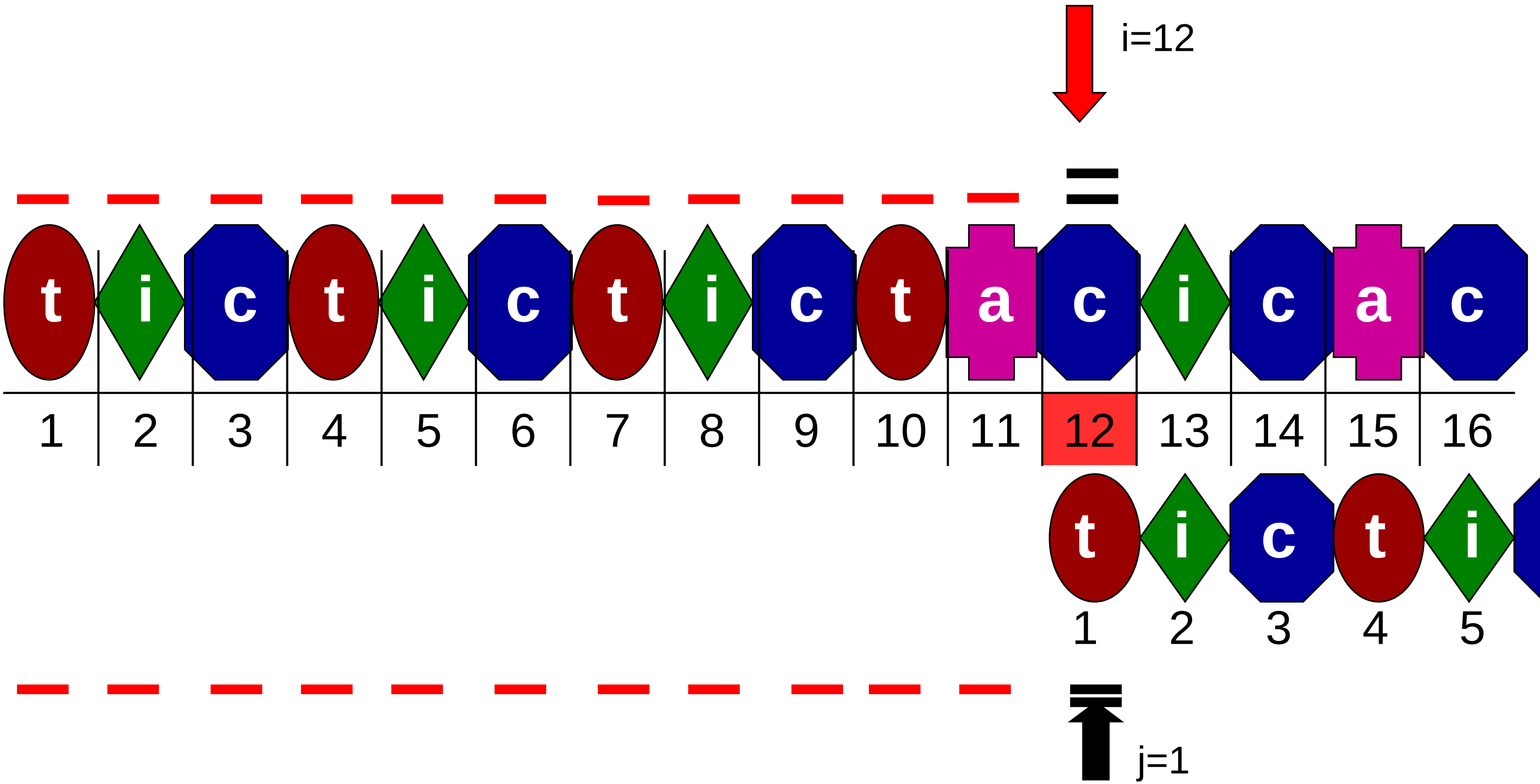
KMP search: no matches at all



0	0	0	1	2	3

P[1] does not match T[11]. We did not match any characters, so we advance i and reset j, starting a new alignment at T[12] with P[1] (as we would do without KMP)

KMP search: overlap 0



0	0	0	1	2	3

etc…

KMP– in “English”

```
T:= 'ticticticttactictictic'
P:= 'tictic'
```

```
N:= len(T)
M:= len(P)
```

```
ol:= [0, 0, 0, 1, 2, 3]
manually precomputed overlap
function for P
```

Setup pointers i and j to point to the current character of T and P respectively

DO

Advance both pointers as long as T[i] matches P[j]

If you advanced all M characters (j=M)

Report occurrence of P in T (at position i-M)

Use an overlap function ol(M) to compute pattern shift

If j≠ M and the next characters T[i] and P[j] does not match:

See how many characters matched - 3 cases:

1. matched 0 characters: advance i, restart j=1 (as we would do without KMP)

2. ol(j-1) = 0. Previous match does not help with alignment,

so we need to start comparing P[1] with T[i] without advancing i

3. ol(j-1)>0. Compute pattern shift and continue comparing from the next j

UNTIL i < N

Full Pseudocode (zero-based)

```
matches: = empty list

i: = 0  # current position in T
j: = 0  # current position in P

while i is within bounds
         loop through both i and j as long as characters T[i] and P[j] match

         if matched all M characters
            add position (i - M) to matches

         if no characters matched
            advance i to the next position in T

         else if some characters matched
            consult the overlap function for the matched prefix of P
            if overlap = 0
                  we have no information about characters in T
                  we restart j, and continue matching from the same T[i]
            else:
                  skip characters in P according to OF

return matches
```

KMP algorithm: time complexity

Theorem: The number of character comparisons in the KMP algorithm is at most $2N$

Proof

- Divide the algorithm into compare/shift parts. Let a single phase include the comparisons done between 2 successive shifts. We see that during 2 successive 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 $2N$

Worst-case example – iterations 1,2

Counting number of times the character is accessed

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

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

Worst-case example – iteration 4

1	1	1	1	4					
a	a	a	a	b	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*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

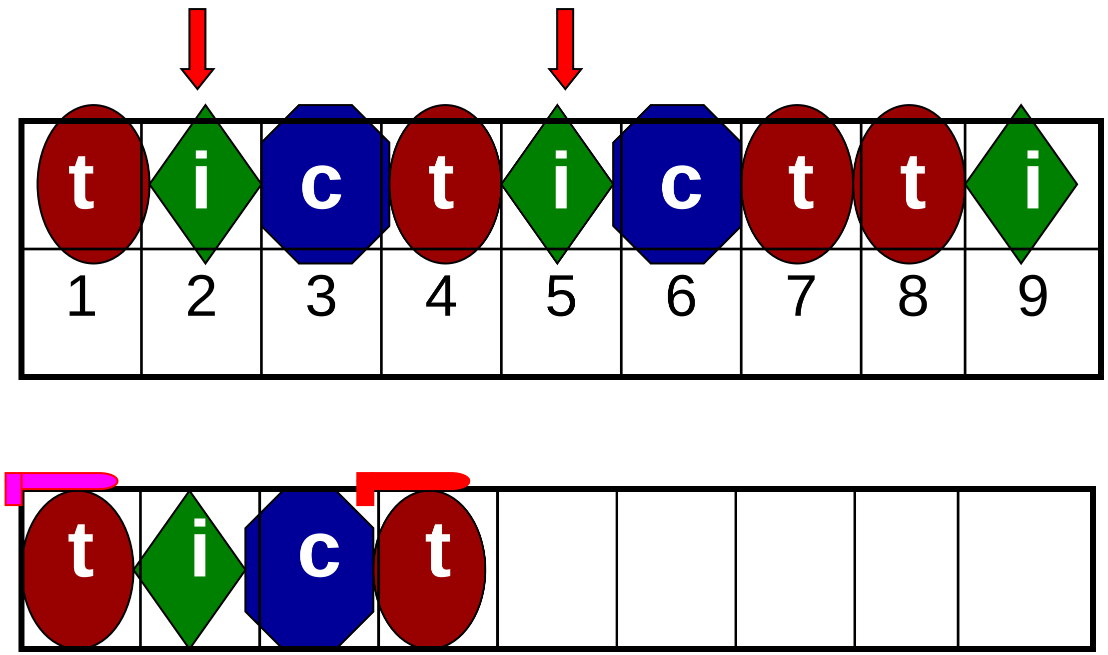
Readings

- http://en.wikipedia.org/wiki/Knuth-Morris-Pratt_algorithm
- http://www.ics.uci.edu/~eppstein/161/960227.html

Overlap function computation in time O(M)

Optional material

How to compute the OF function



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

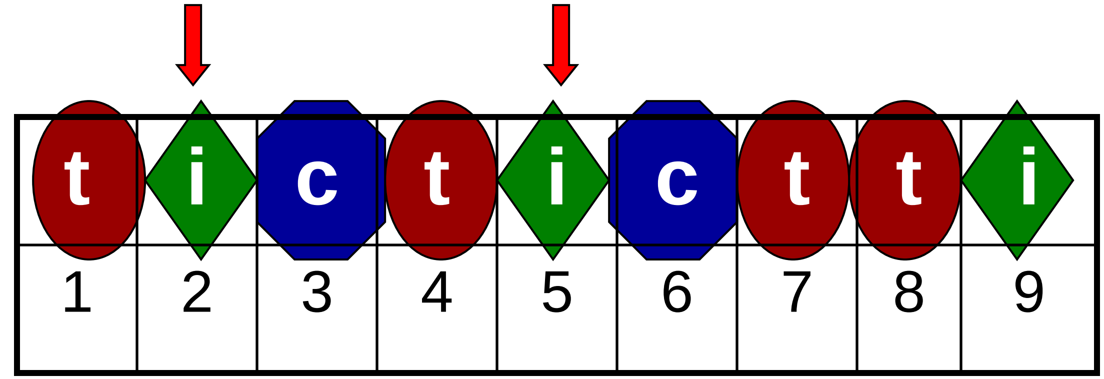
The easy case:

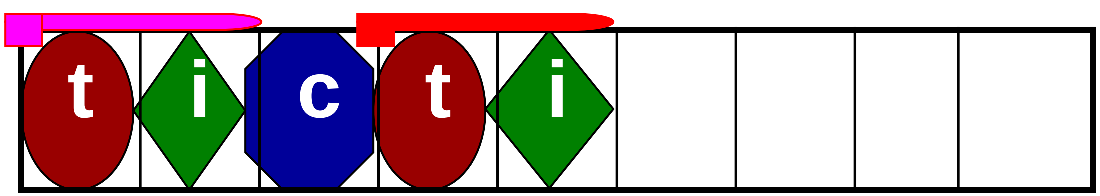
if we have OF(j-1), and the characters

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

Then we just increase 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				

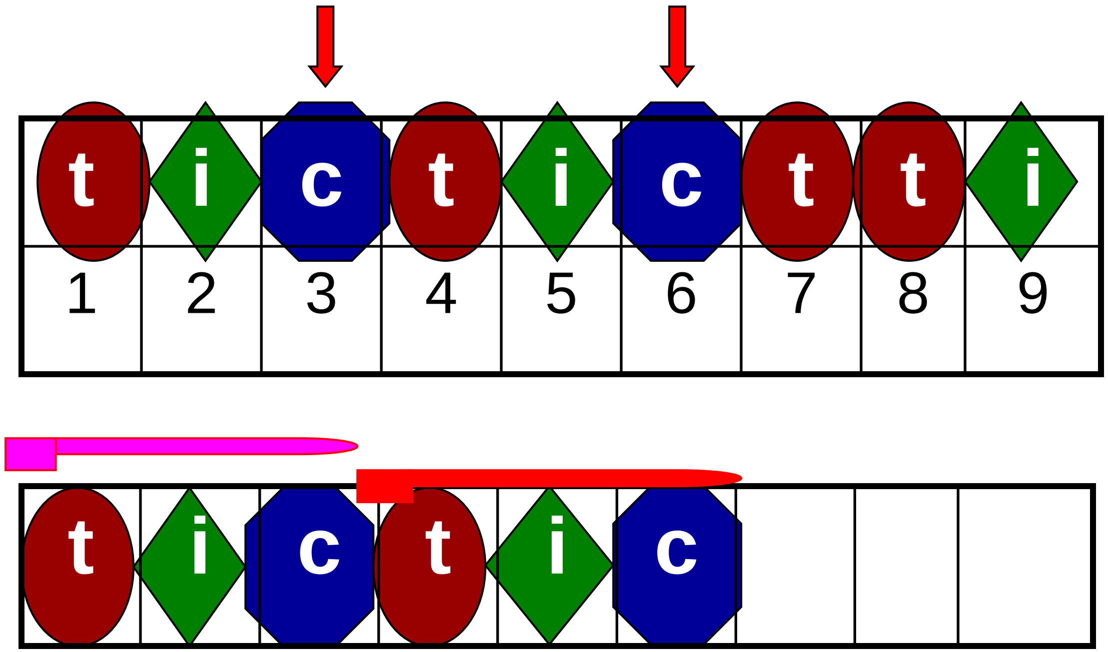
The easy case:

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			

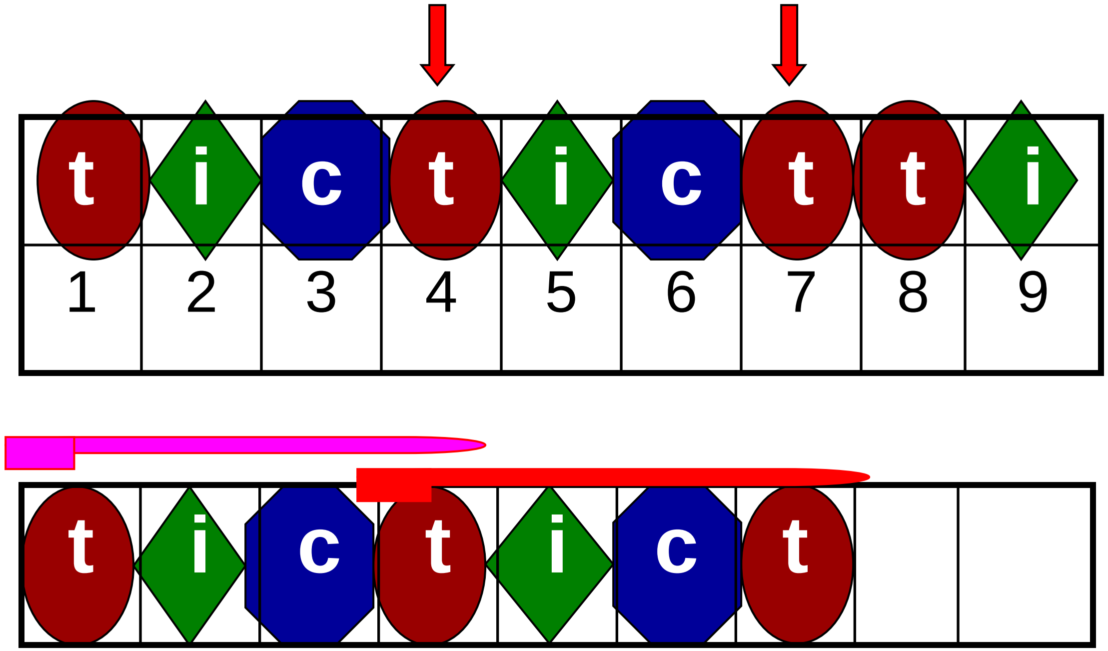
The easy case:

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		

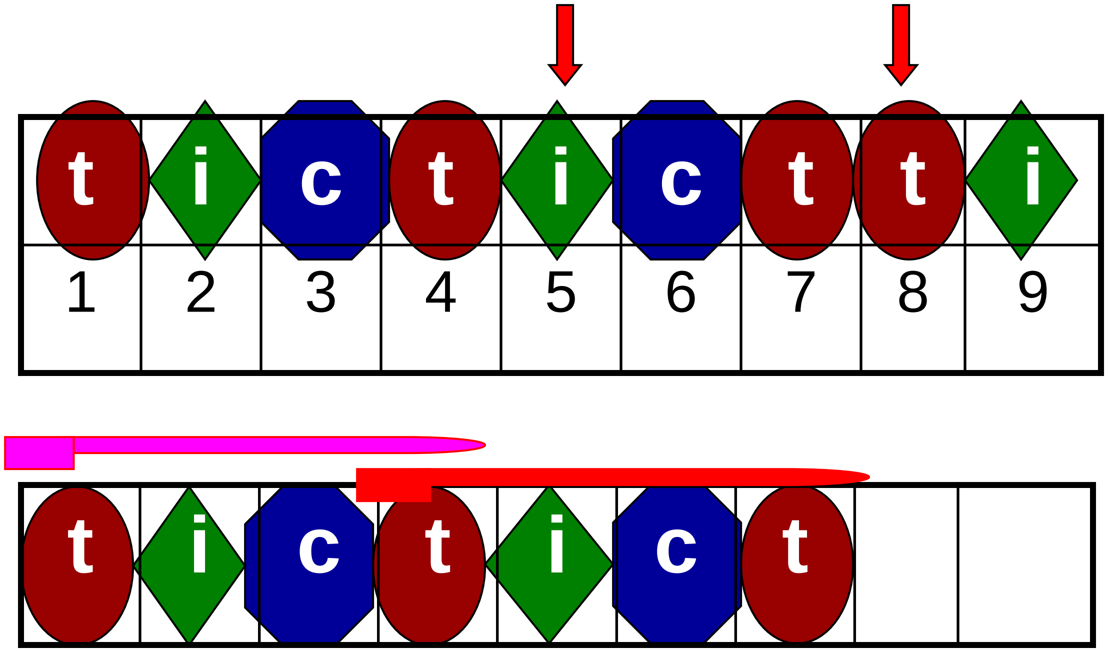
The easy case:

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		

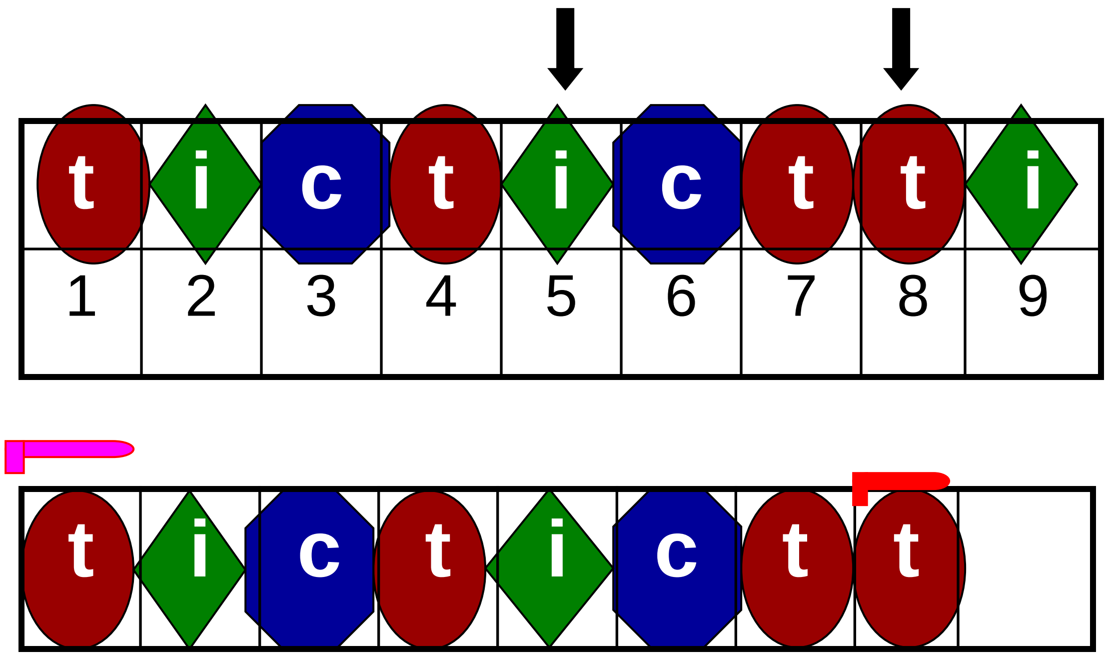
The general case:

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

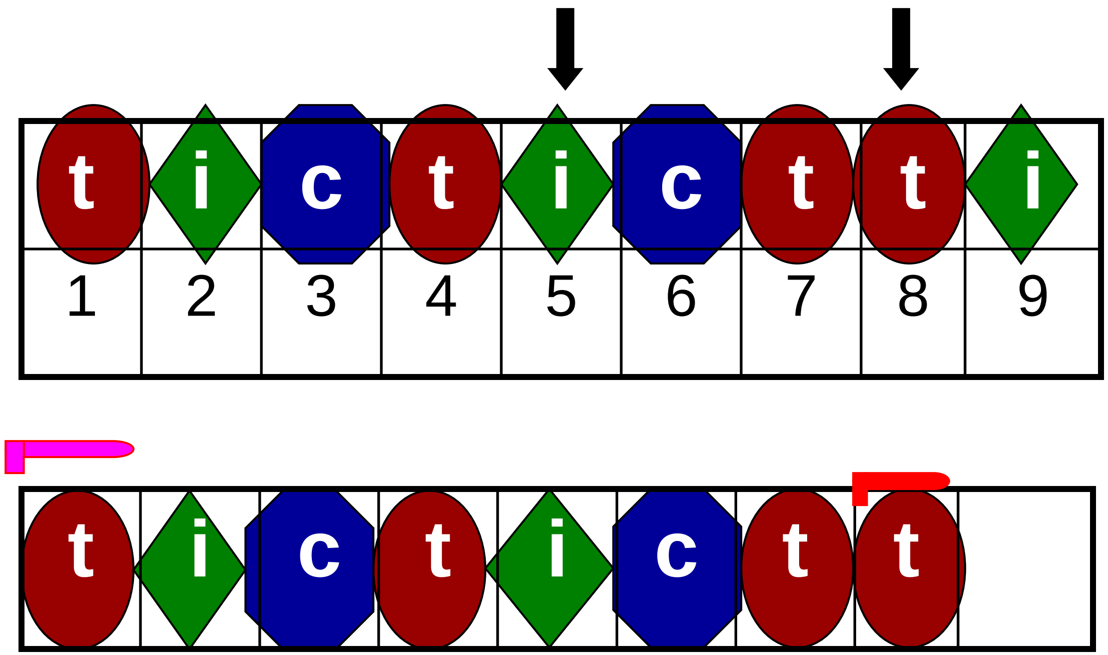
then OF(j) is less than OF(j-1)

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



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

The general case:

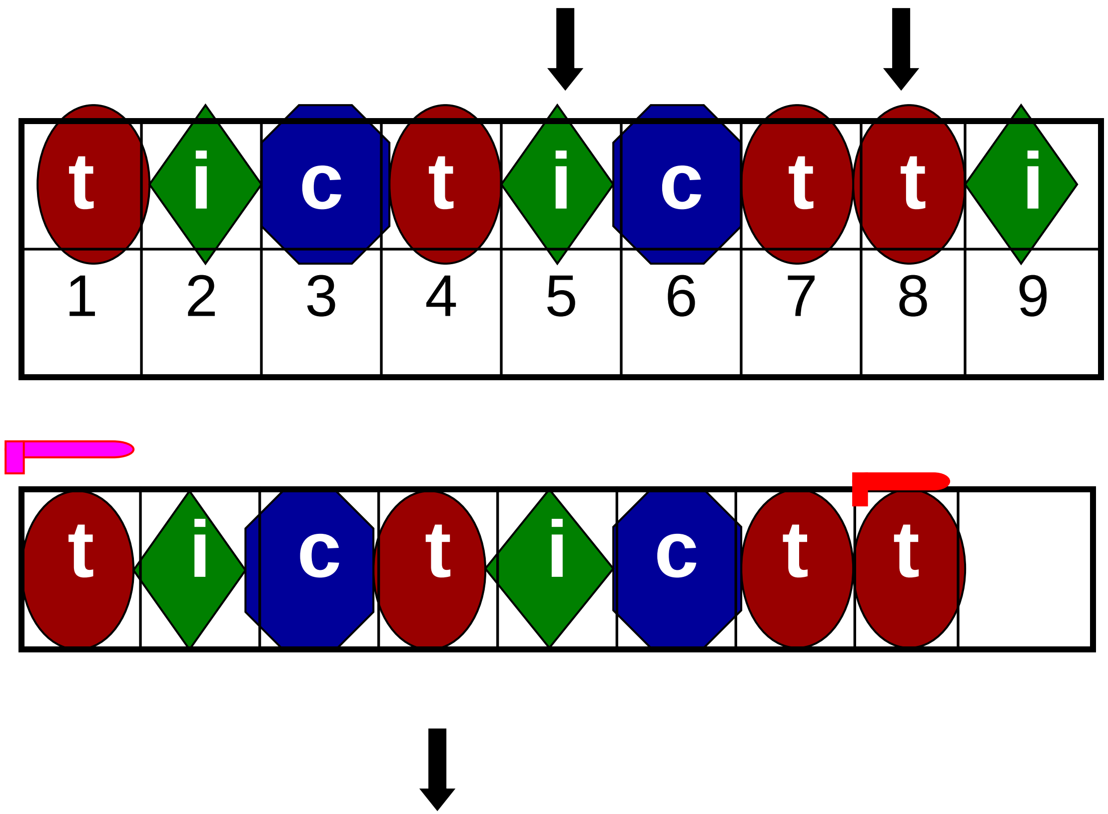
If the characters

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

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)] do not match

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

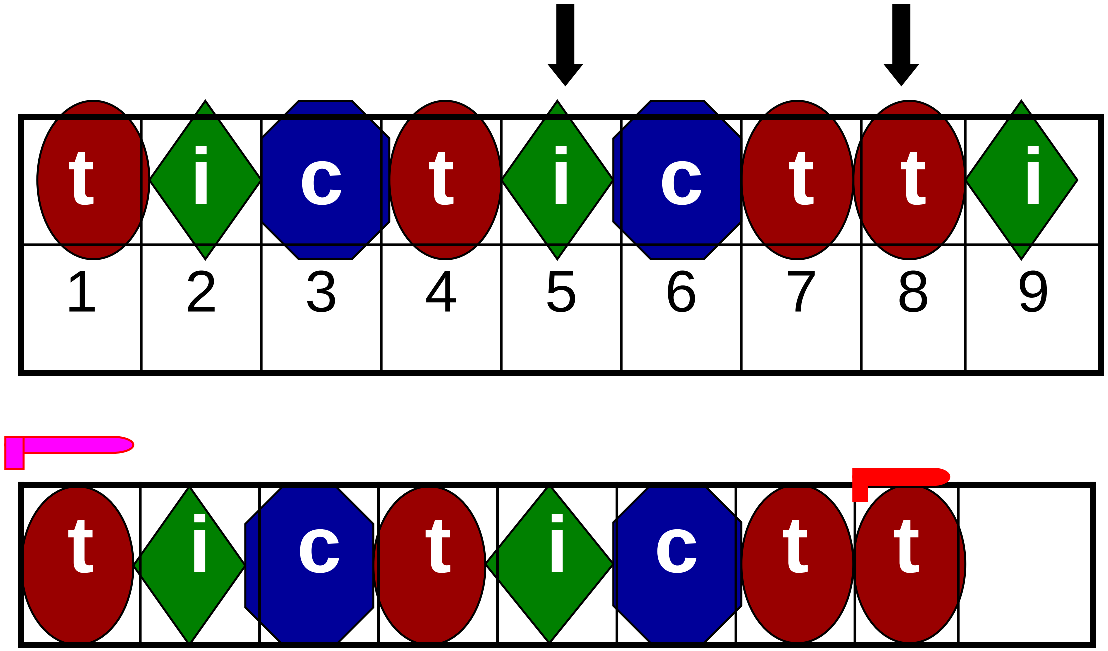
We look at v=OF(j-1) and check again the next character

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]≠P[8]

v=OF(4)

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:

v=OF(4)

P[1]=P[8], thus
OF(8)=OF(1)+1=1

Why is this correct

t	**i**	**c**	**t**	**i**	**c**	**t**	**t**	**i**
1	2	3	4	5	6	7	8	9

t	**i**	**c**	**t**	**i**	**c**	**t**	**t**	
1	2	3	4	5	6	7	8	9

1	2	3	4	5	6	7	8	9
0	0	0	1	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 new overlap we are looking for is less than these 4 letters

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

Why is this correct

t	**i**	**c**	**t**	**i**	**c**	**t**	**t**	**i**
1	2	3	4	5	6	7	8	9

t	**i**	**c**	**t**	**i**	**c**	**t**	**t**	
1	2	3	4	5	6	7	8	9

1	2	3	4	5	6	7	8	9
0	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 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

Overlap function – pseudocode (0-based)

```
ol: = table of size M with all zeroes

ol[0]: = 0   # first overlap is always 0

for pos from 1 to M -1:
    prev_overlap: = ol[pos - 1]

    if P[pos] = P[prev_overlap]: # if next character is the same
        ol[pos]: = prev_overlap + 1 # overlap becomes bigger

    else: # the suffix does not extend previous suffix
        while P[pos]!=P[prev_overlap] and prev_overlap ≥ 1:
        # try extend a smaller prefix - based on P [ol[pos-1]]
            prev_overlap: = ol[prev_overlap - 1]

            if P[pos] = P[prev_overlap]:
                ol[pos] = prev_overlap + 1
        # if we did not find any overlap to extend
        # then ol[pos] remains 0

return ol
```

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 *M* times
- in addition, the value of *v* is decreasing inside the *while* loop, but since *v* is never less than zero, the total number it is decreasing can not be more than the number it is increasing, therefore is bounded by *M* too.

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

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	10	11	?

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	10	11	?

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…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	10	11	?

Let us check if this value is also the maximum overlap for the substring P[1…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	10	11	**3**

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

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

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

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

Check your understanding: 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	**?**

Solution step 1

	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
OL	0	1	2	0	1	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	8	9	**?**

Solution step 2

	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
OL	0	1	2	0	1	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	8	9	**?**

Solution step 3

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