

Pattern search.
Algorithm by
Knuth, Morris, Pratt (KMP)

Lecture 2

by Marina Barsky

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

“The ultimate rationale behind all purposeful structures and behavior of living things is embodied in the sequence of residues of nascent polypeptide chains . . . In a real sense it is at this level of organization that the secret of life (if there is one) is to be found.”

We use pattern search for:

- Finding overlaps during sequence assembly
 - Finding unique sequences used to map the positions of the fragments in the genome
 - Finding promoter sequences that signal beginning of a coding region
 - Subroutine for more complex string algorithms
-

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*

- $S[i\dots j]$ is a contiguous *substring* of S starting at position i and ending at position j of S
- $S[1\dots j]$ is a *prefix* of S starting at position **1** and ending at position j
- $S[i\dots N]$ 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

- $S[1\dots j]$ is a *prefix* of S starting at position 1 and ending at position j
- $S[i\dots N]$ is a *suffix* of S starting at position i and running till N
- $S[i\dots j]$ 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

- $S[1\dots j]$ is a *prefix* of S starting at position 1 and ending at position j
- $S[i\dots N]$ 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?

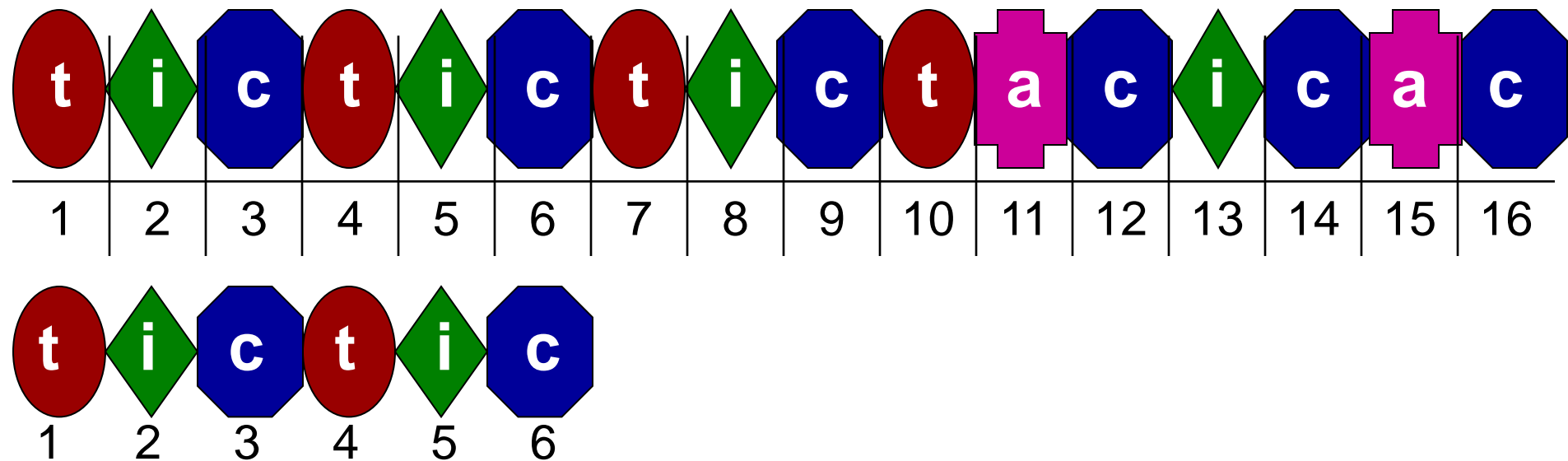
Is Prefix 0 a proper prefix?

Is Suffix 1 a proper suffix?

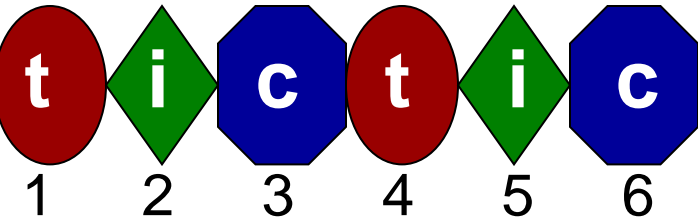
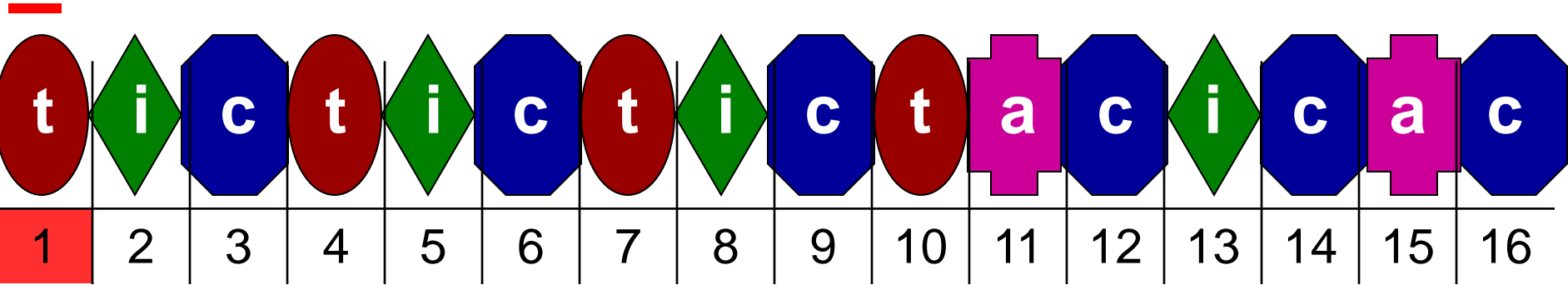
Pattern matching problem

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

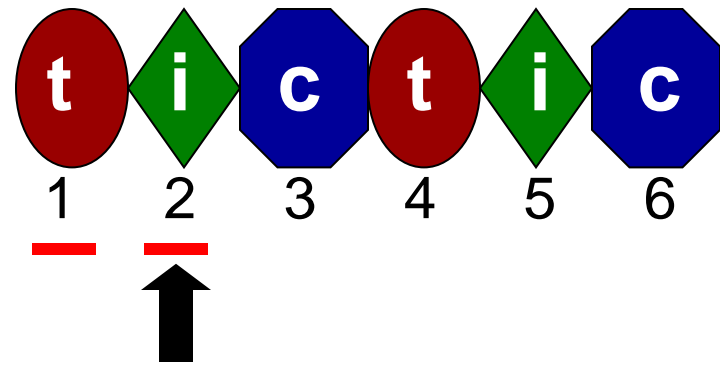
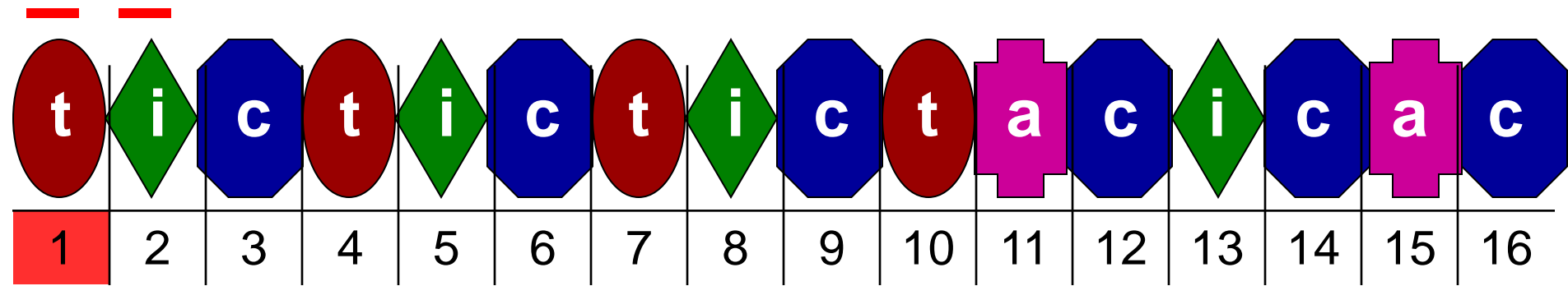
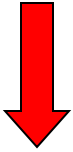
Naïve exhaustive search



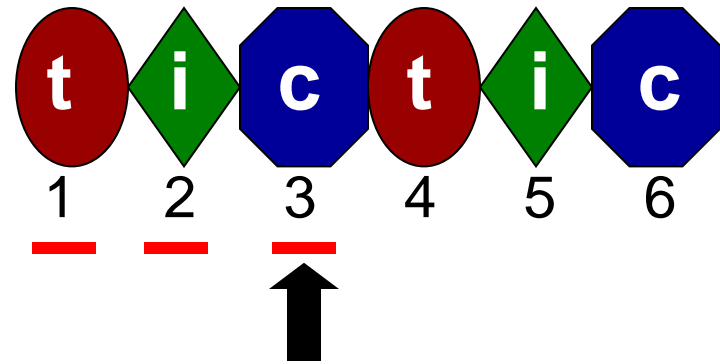
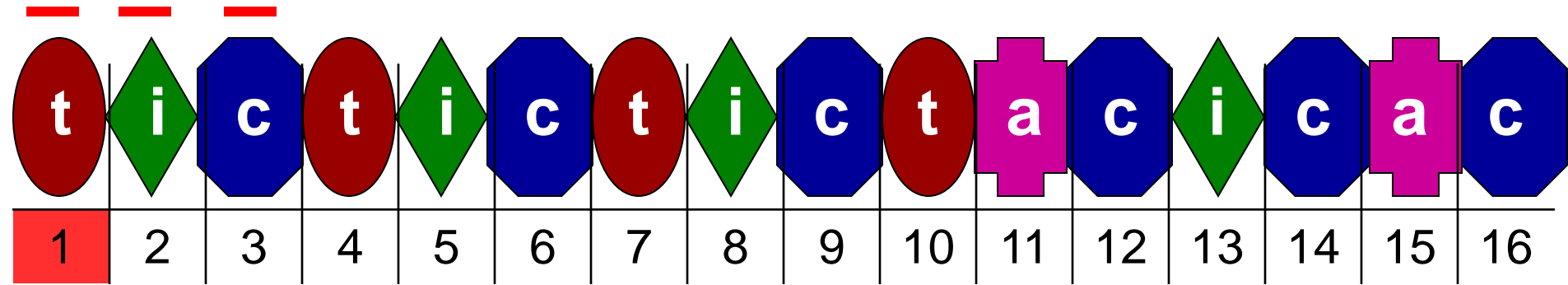
Naïve exhaustive search



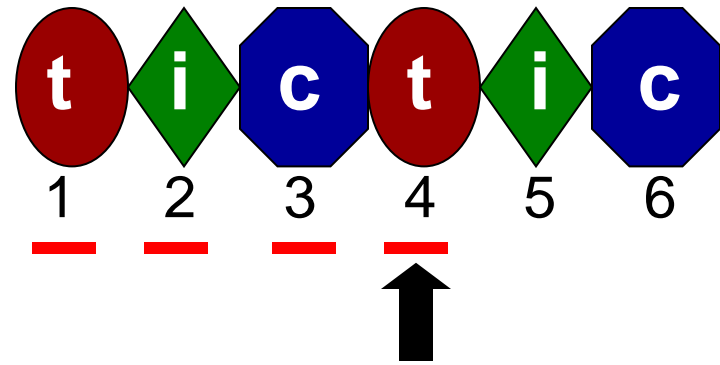
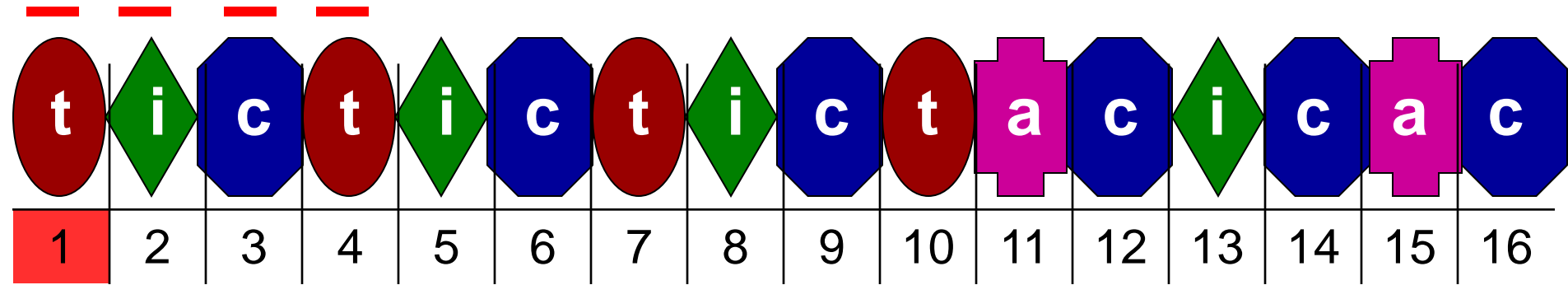
Naïve exhaustive search



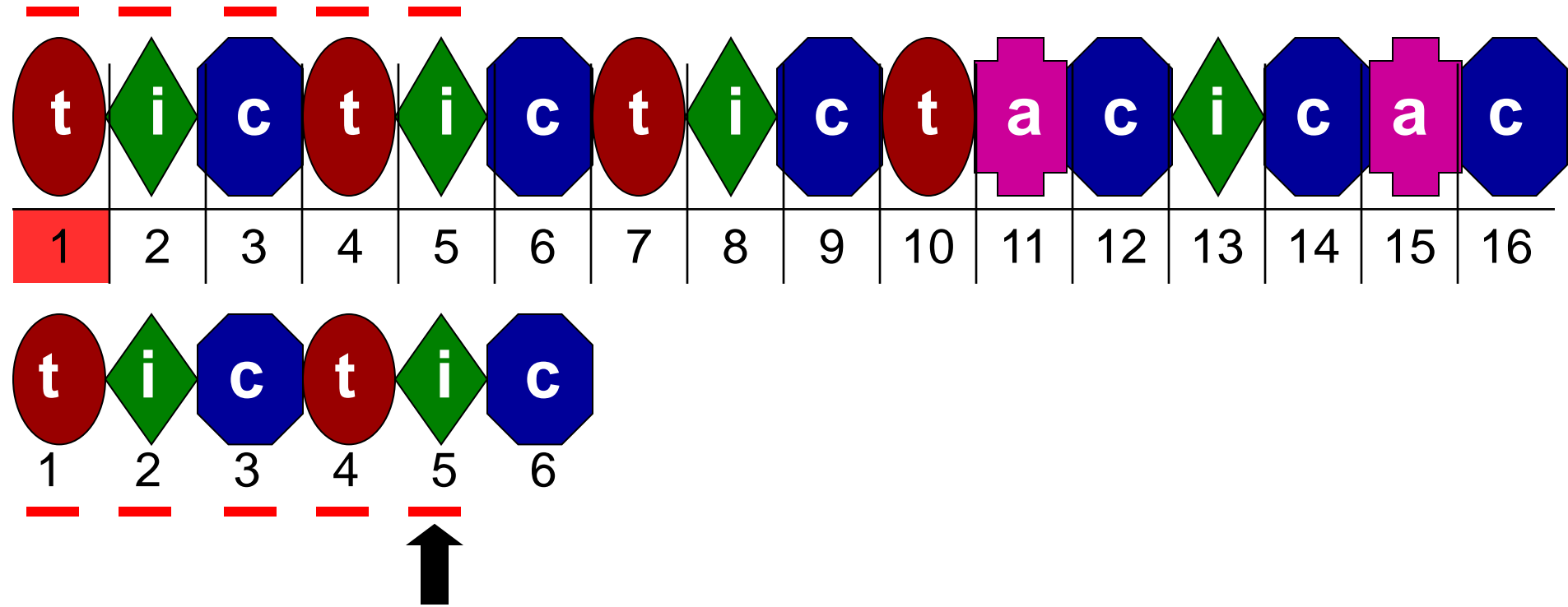
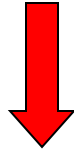
Naïve exhaustive search



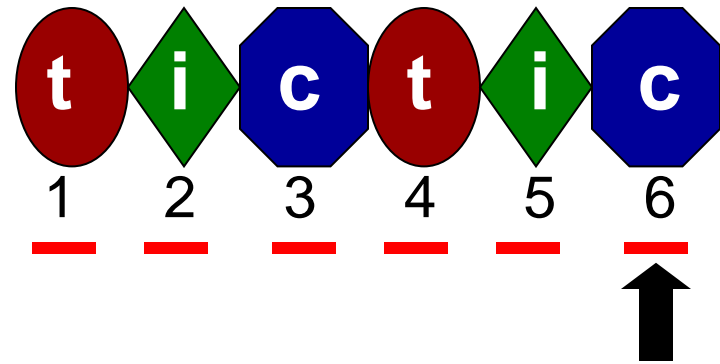
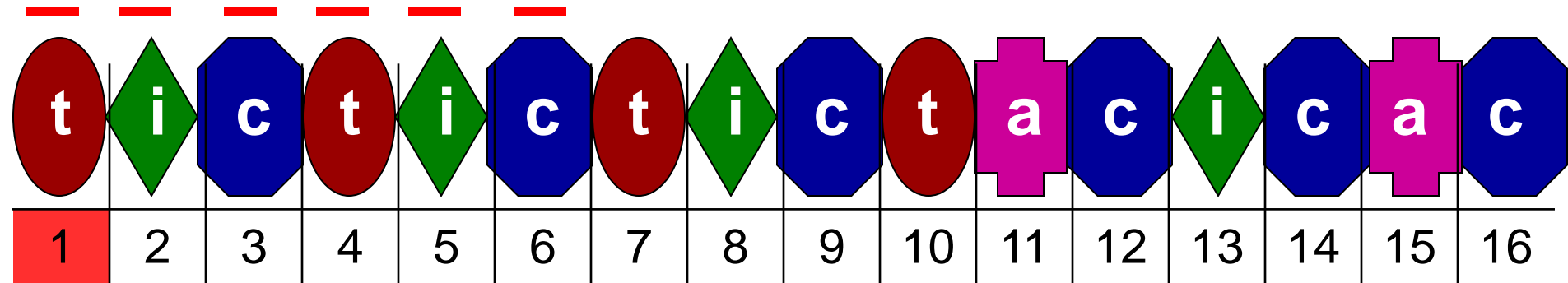
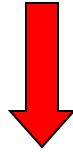
Naïve exhaustive search



Naïve exhaustive search



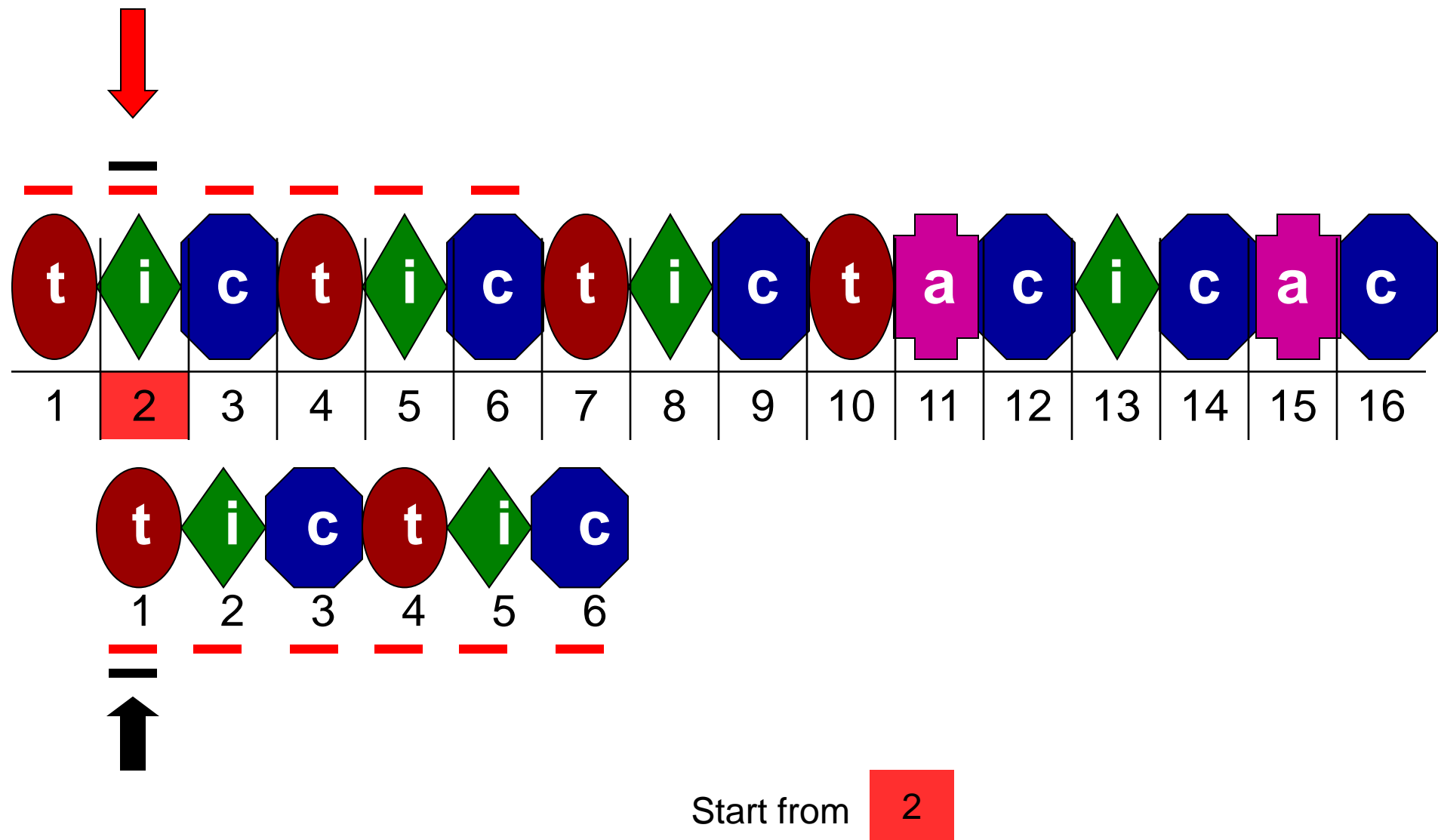
Naïve exhaustive search



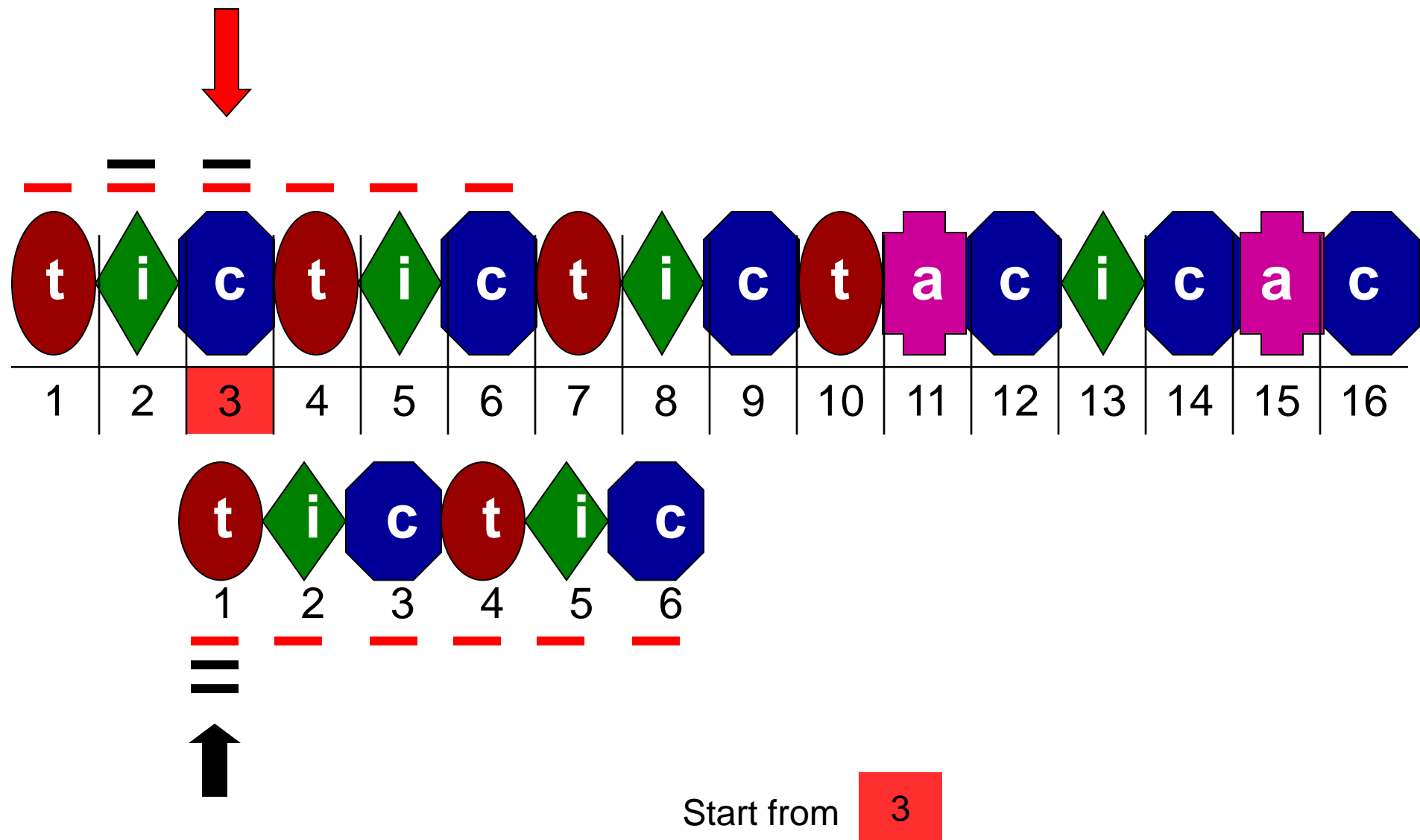
Report

1

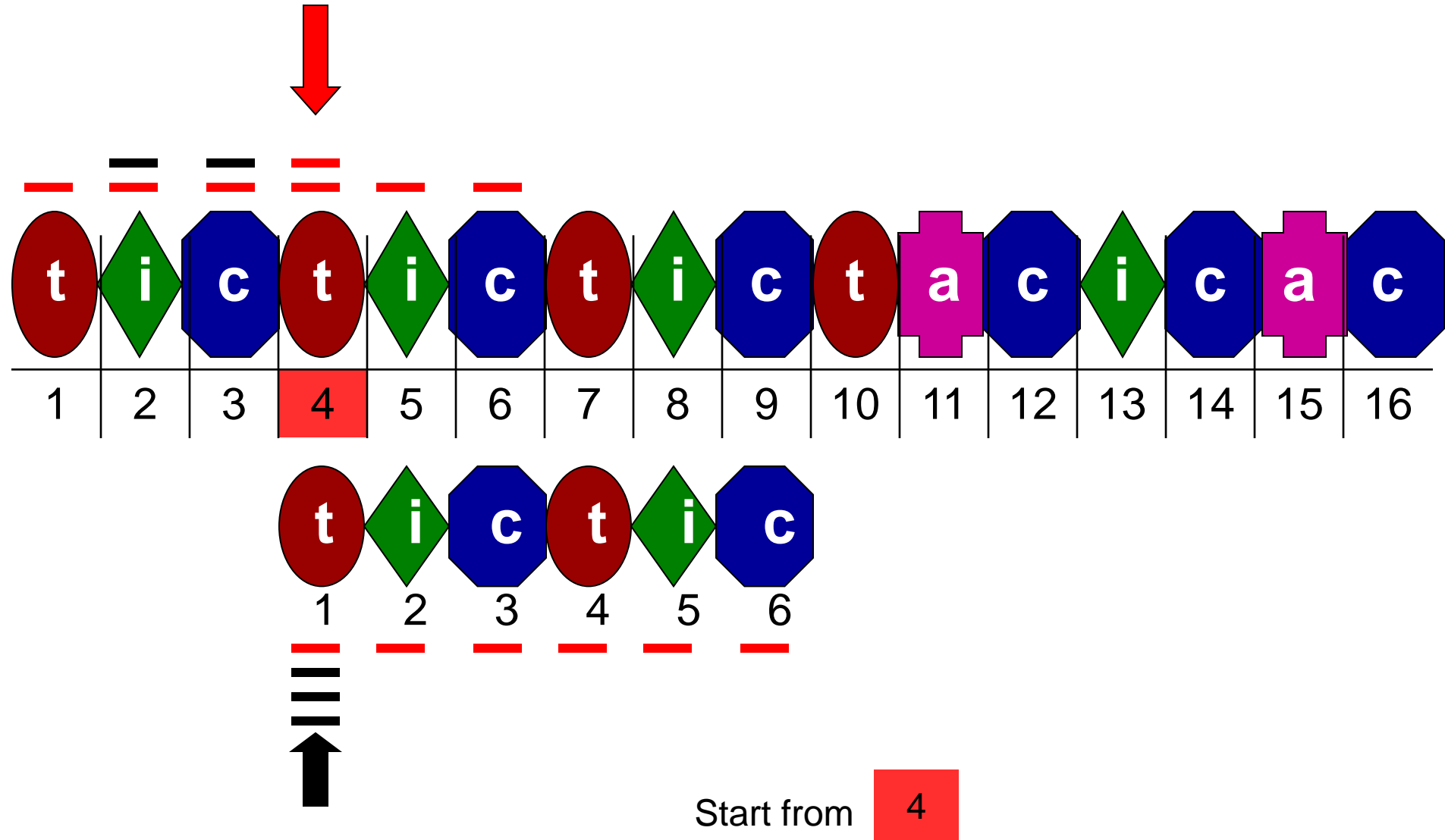
Naïve method – what next?



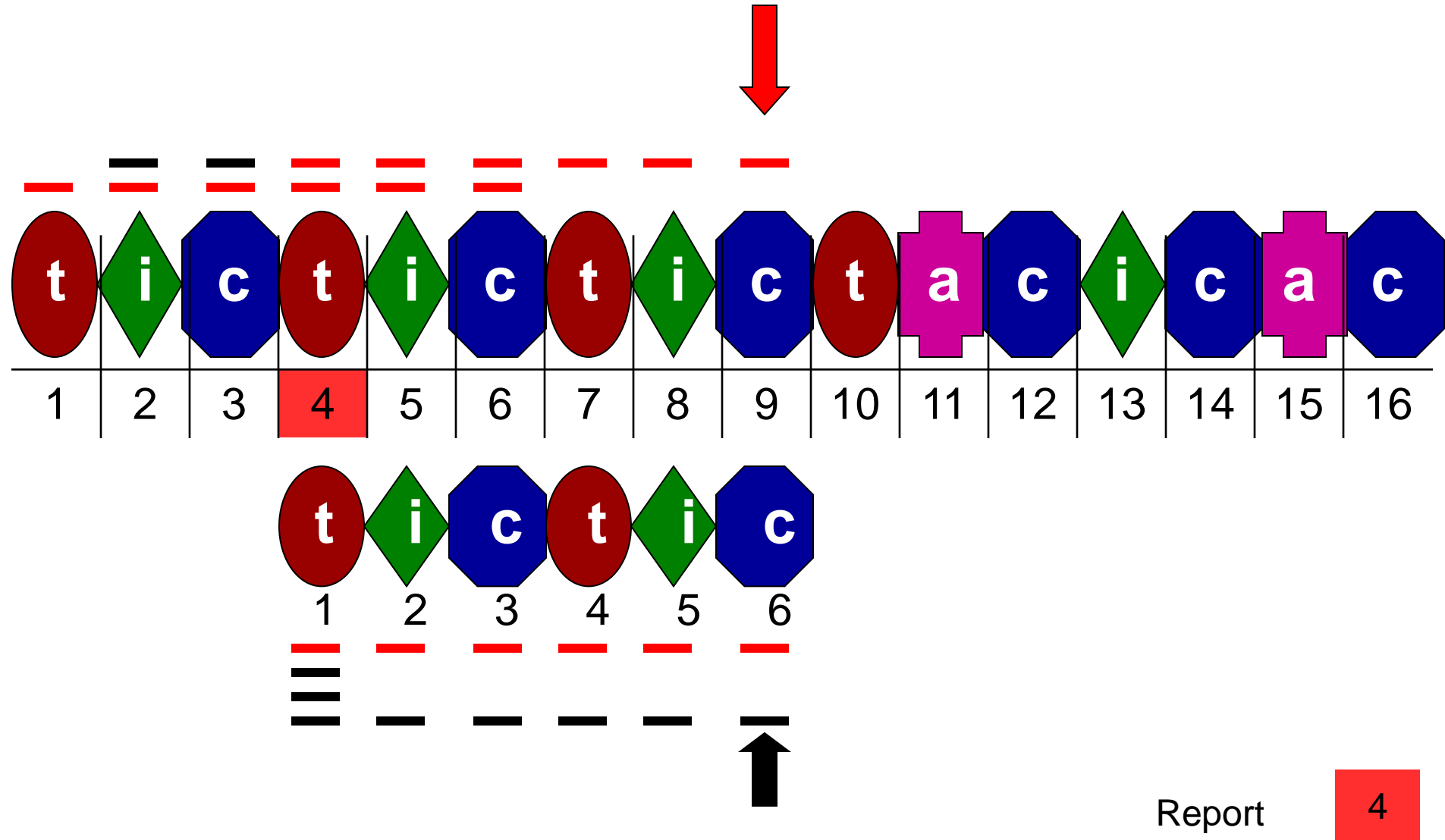
Naïve method – what next?



Naïve method – what next?



Naïve method – continue...



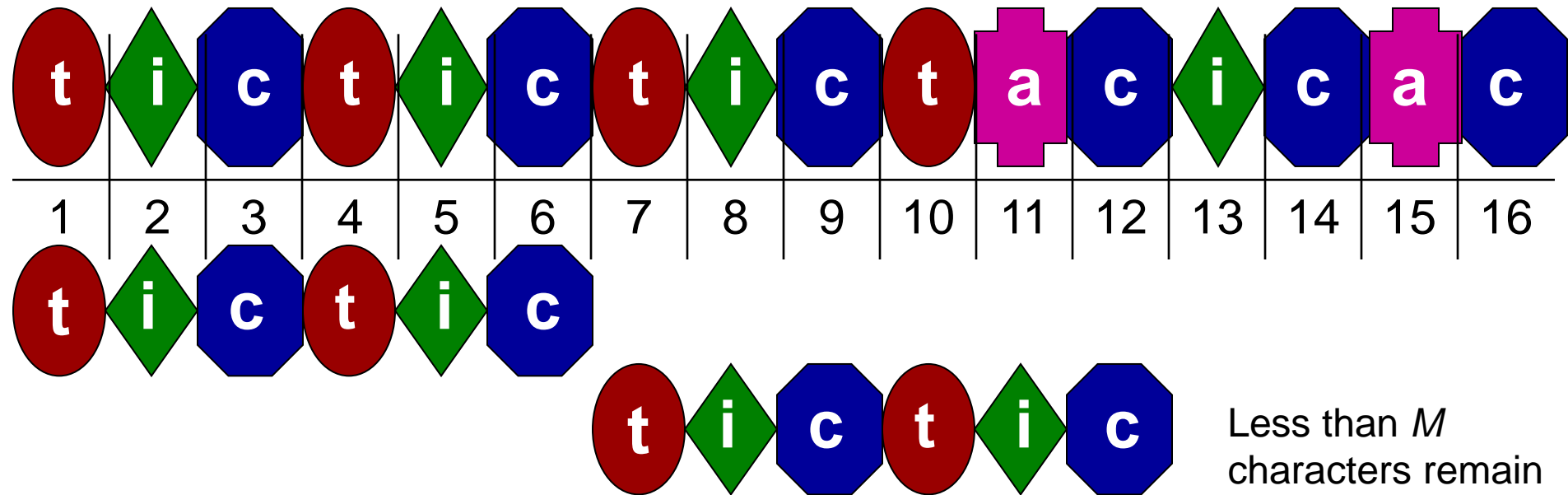
Naïve method – time complexity

- 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 , we 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

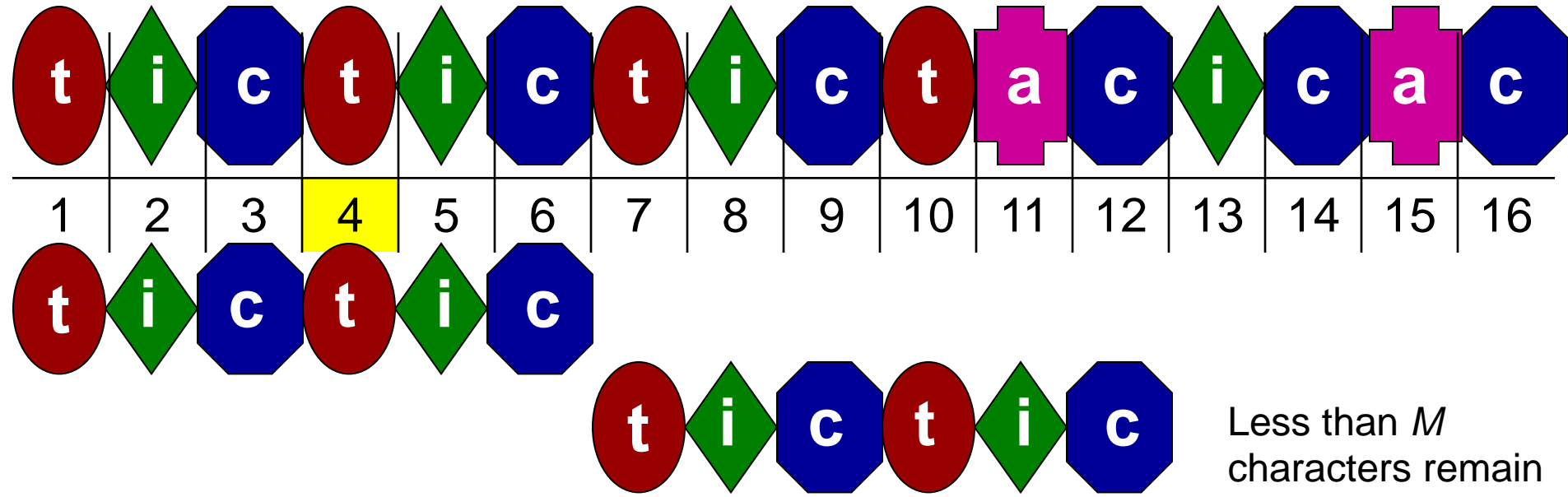
- Let the length of the pattern $M=100$
- A standard fetching time from sequential RAM is 358 MB values per second ([ref](#))
- If we have 10 genomic sequences 3GB each, then we need to search through the text of a total size $N=3 \times 10^{10}$, which can be sequentially accessed in approximately 3×10^8 values per second. We will spend 100 seconds on a linear time algorithm, but for the naïve $O(MN)$ algorithm 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

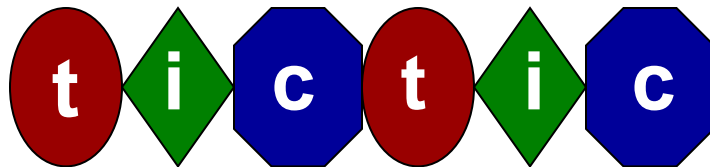


Is this algorithm correct?

Incorrect algorithm



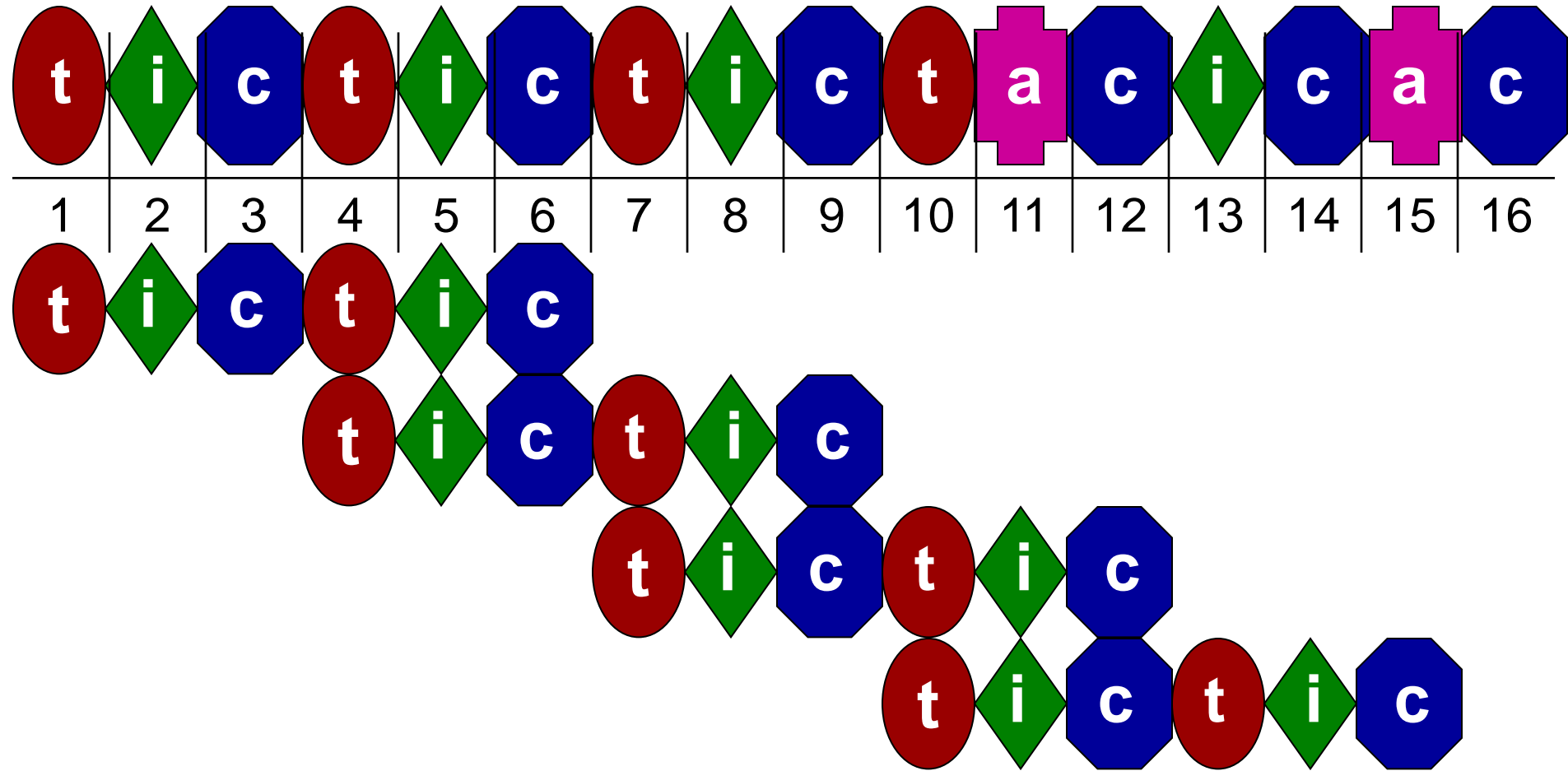
No, we missed an occurrence of P starting at position 4



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!

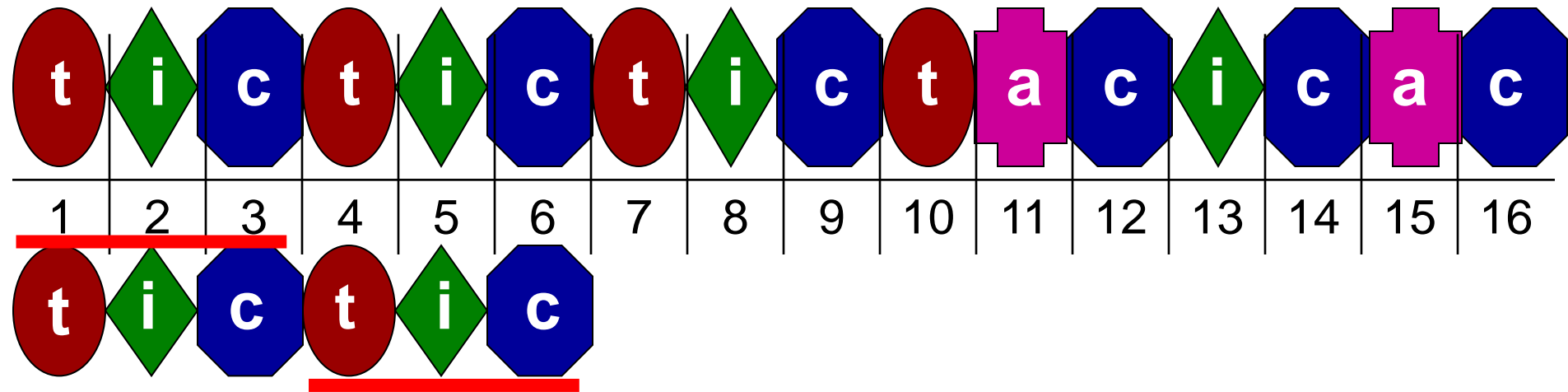
Shifting heuristics

- 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 $P[1\dots k]$ of P).
- From this information we can deduce the place where to start the next comparison.

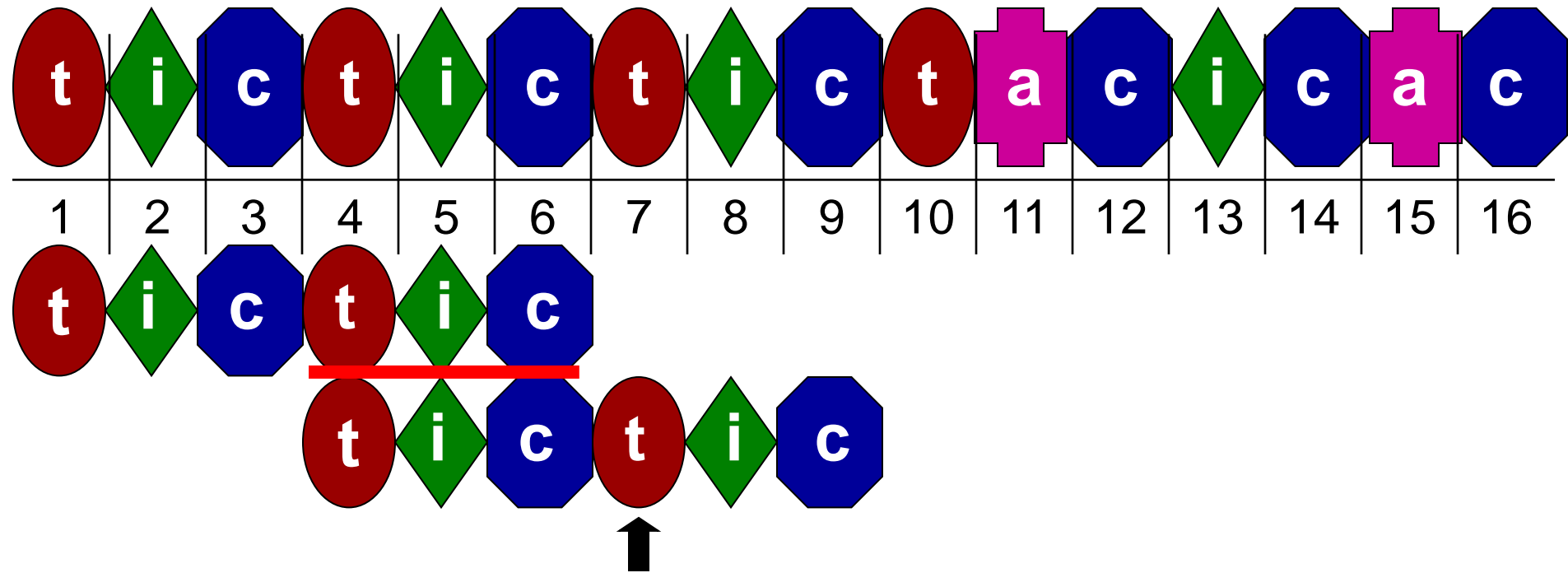
KMP intuition



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

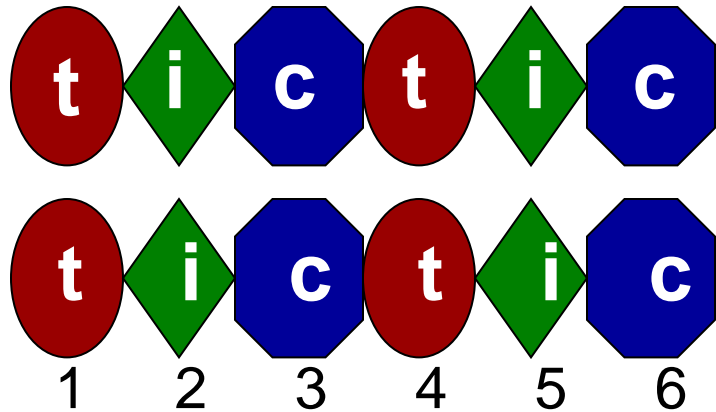


Therefore we can set a start of the next comparison to 3 positions backwards from the current position, and we **don't need to compare the first 3 characters again**, since we know that they match

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

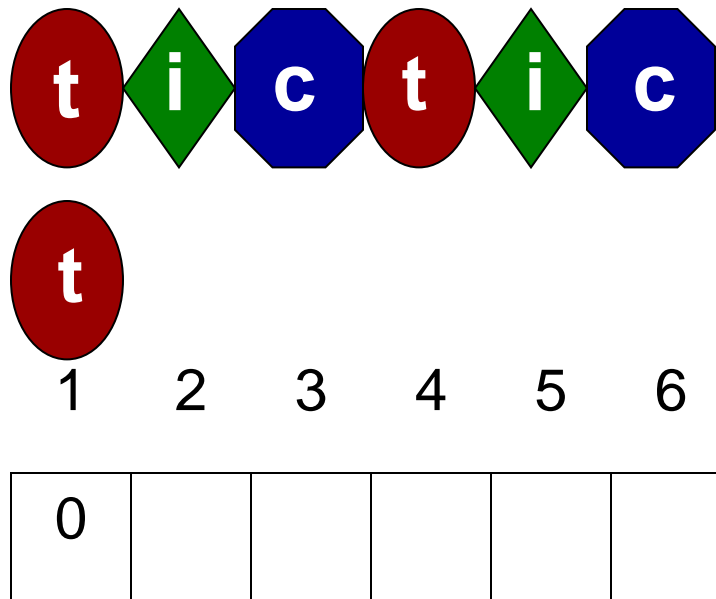


In order to know where to position the start of the next comparison, we need to know the values of an **overlap function** for P , namely:

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 $P[1, j]$.

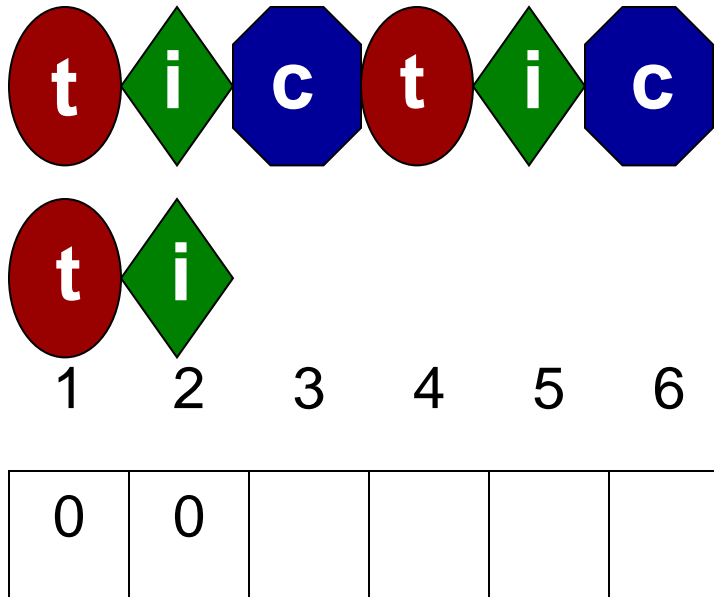
Before we start the search, we need to compute an overlap function for P – we need to **preprocess** pattern P .

KMP intuition – overlap function for P



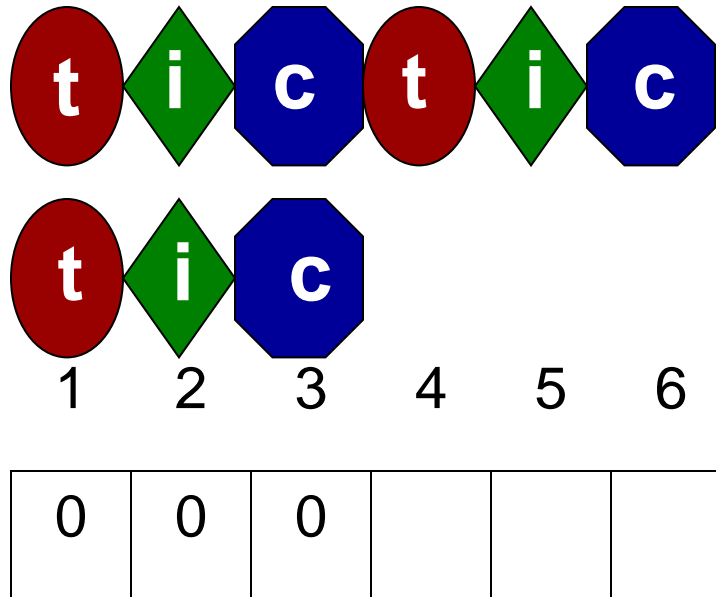
For $j=1$, $OF=0$ (t is not a proper suffix of a substring t , it is the entire t !)

KMP intuition – overlap function for P



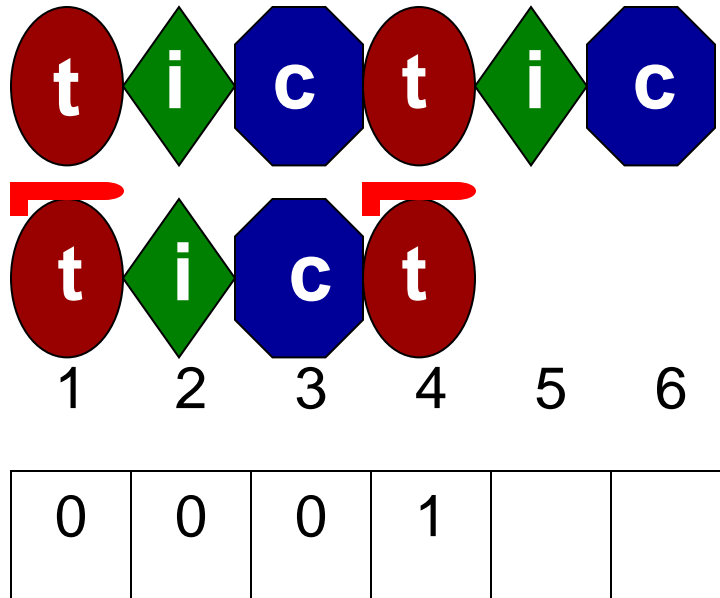
For $j=2$, $OF=0$ (the only proper suffix of ti , the suffix i , does not have overlap with any prefix of ti)

KMP intuition – overlap function for P



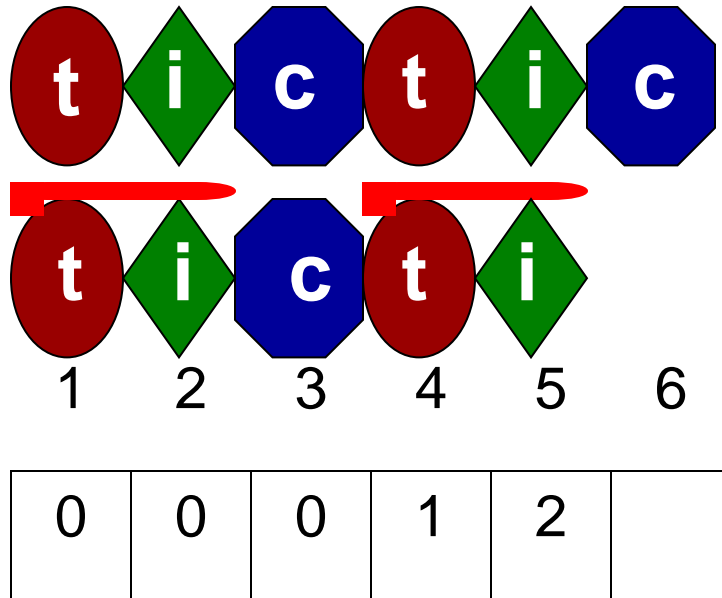
For $j=3$, $OF=0$ (suffixes *ic*, *c* do not have an overlap)

KMP intuition – overlap function for P



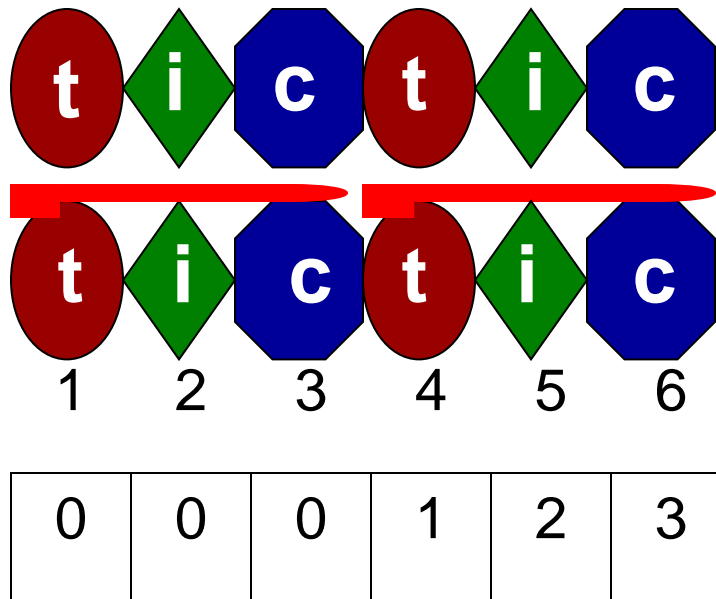
For $j=4$, $OF=1$ (t is a proper suffix of a substring *tict*, and the prefix of P)

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

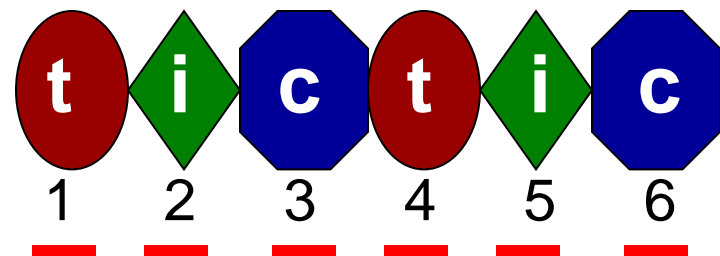
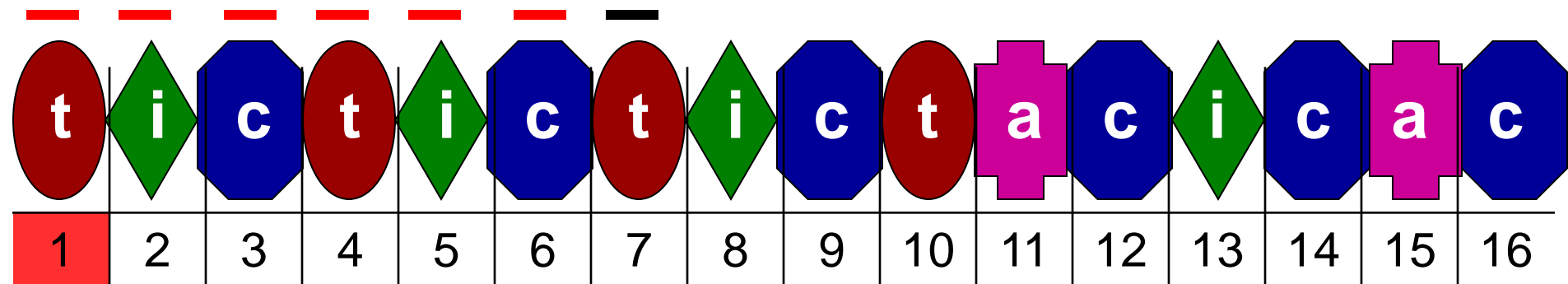
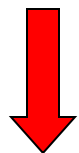


For $j=6$, $OF=3$ (*tic* is a proper suffix of a substring *tictic*, and the prefix of P)

Assume, for now, that the OF values for P are pre-computed

KMP search: match found

$i=7$



$j=7$



Report

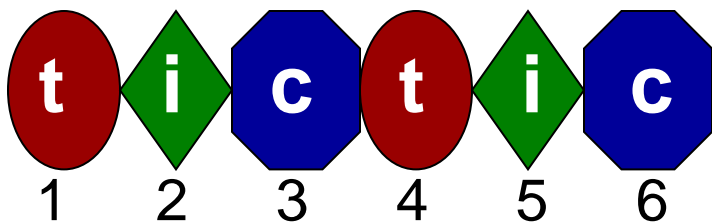
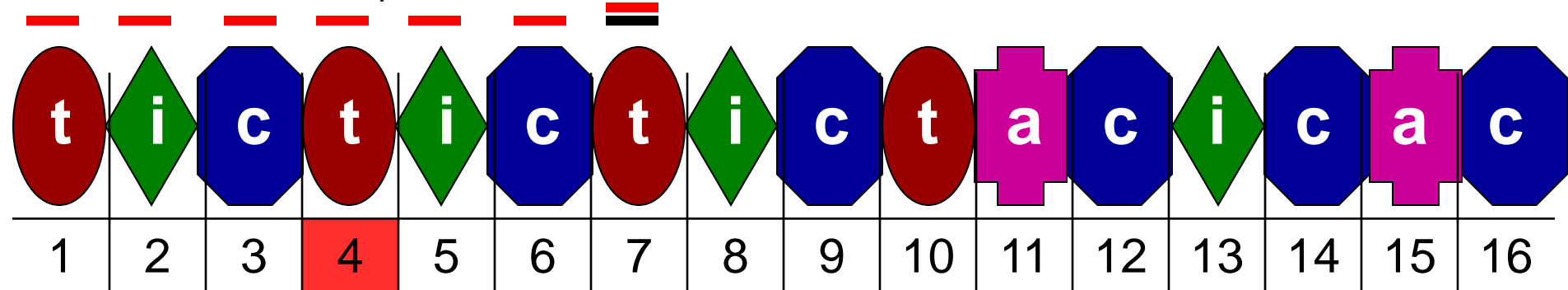
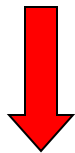
1

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: overlap 3

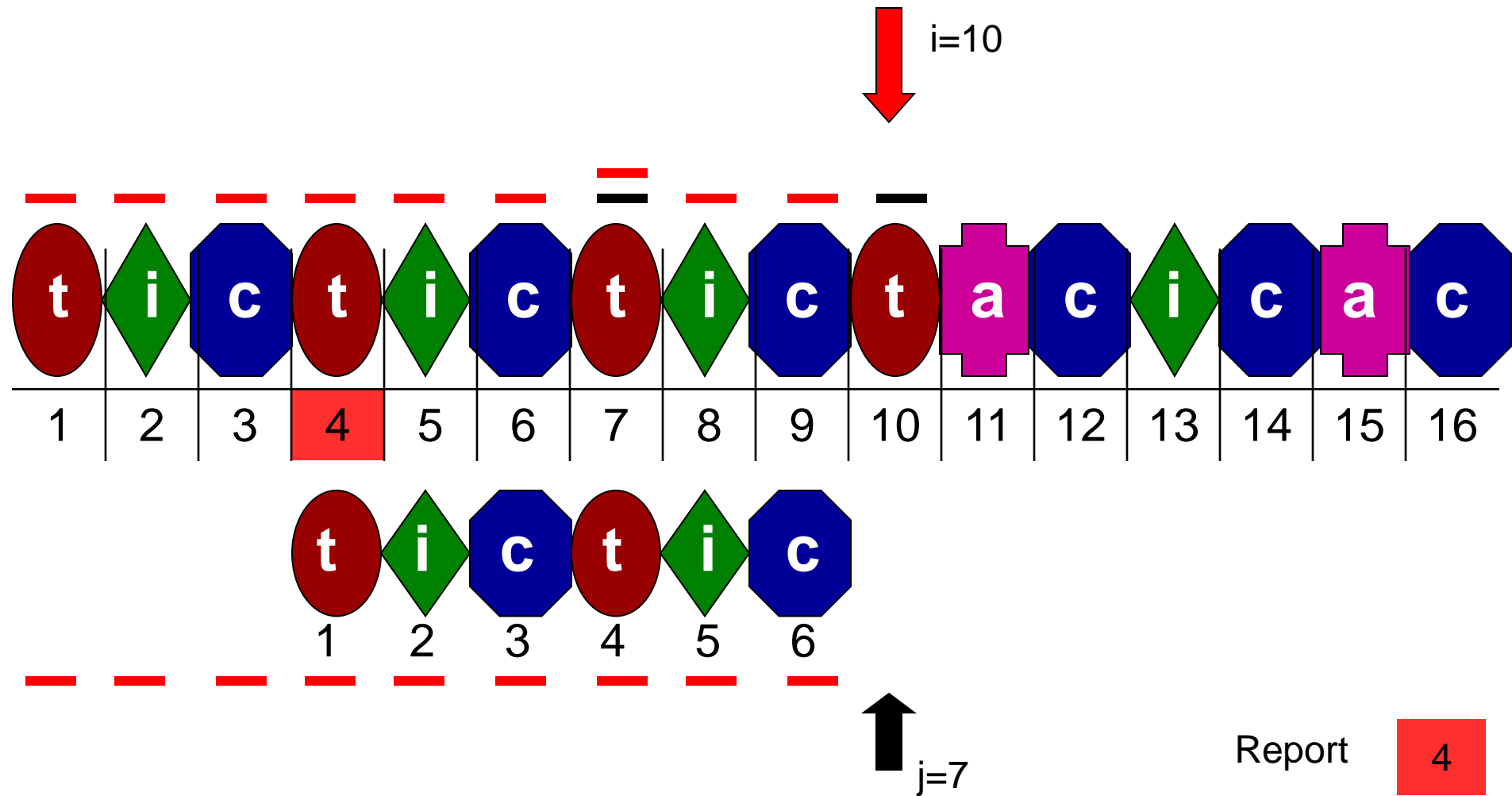
No need to compare these 3 characters, we know that they match – we just compared them



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

Next alignment starts at: $k=4$

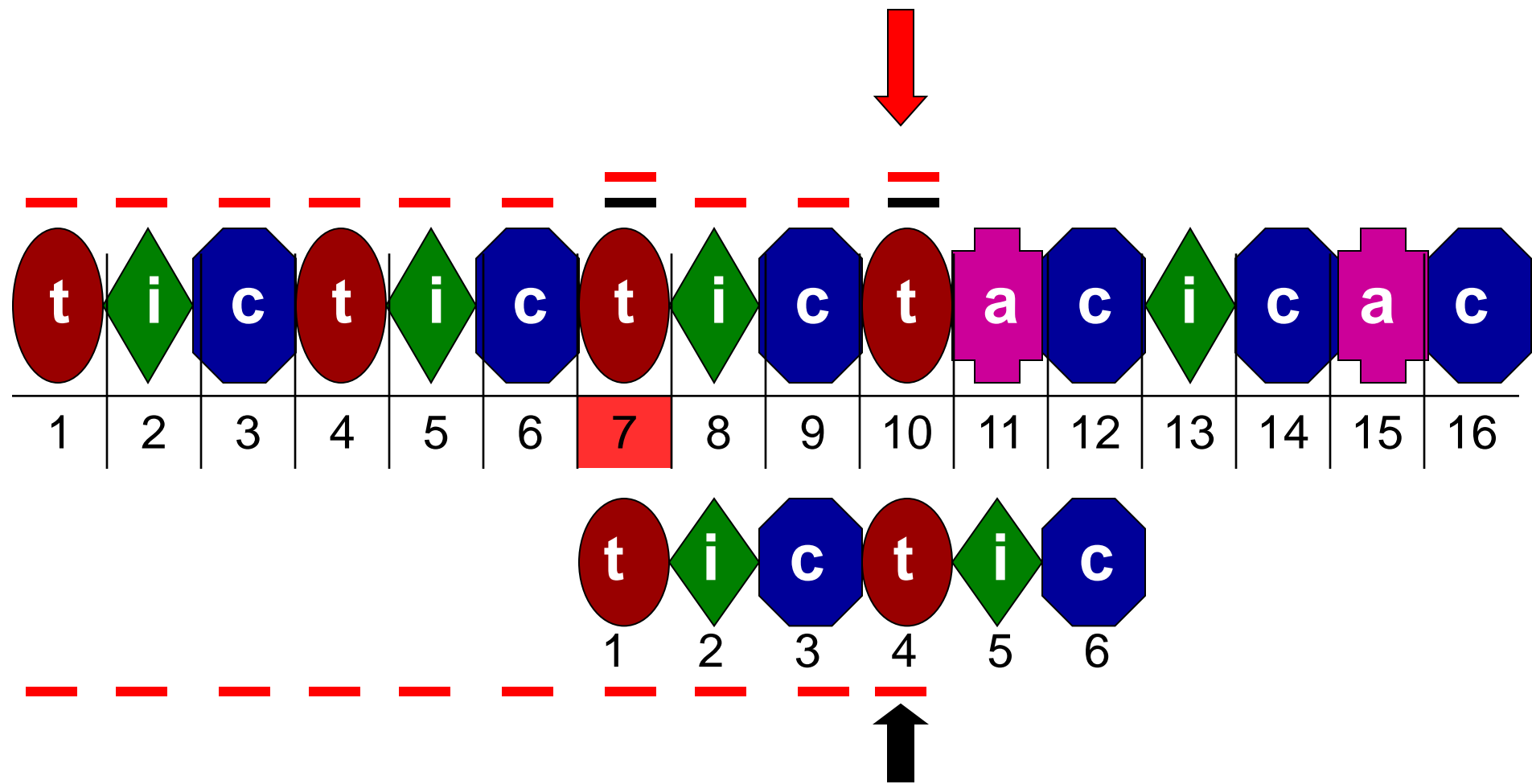
KMP search: overlap 3



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

Consult $OF(6)=3$ it tells how many positions backward from i the next comparison starts: $k=i-OF(j-1) = 10-3=7$

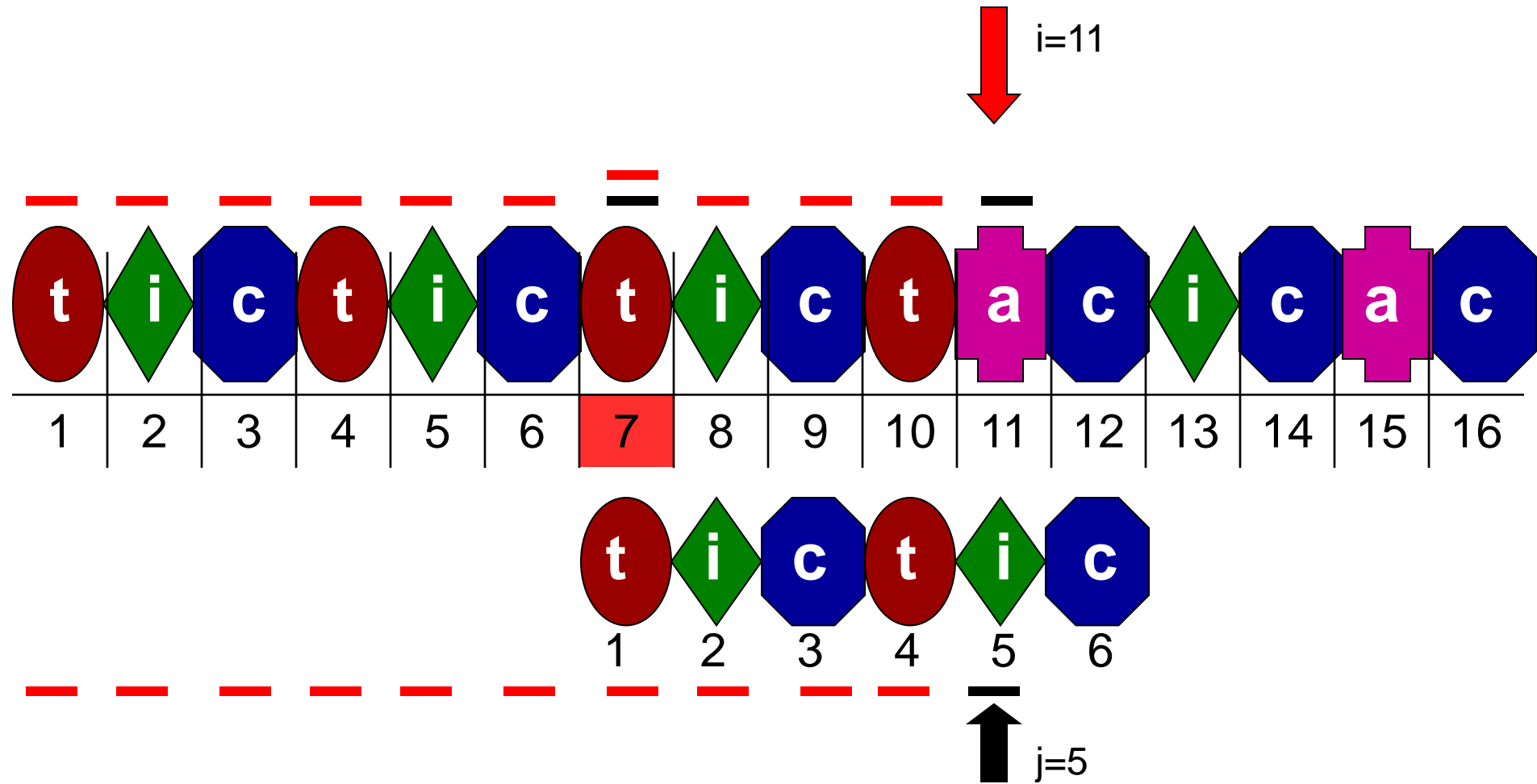
KMP search



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

Continue comparing T[10] and P[4]

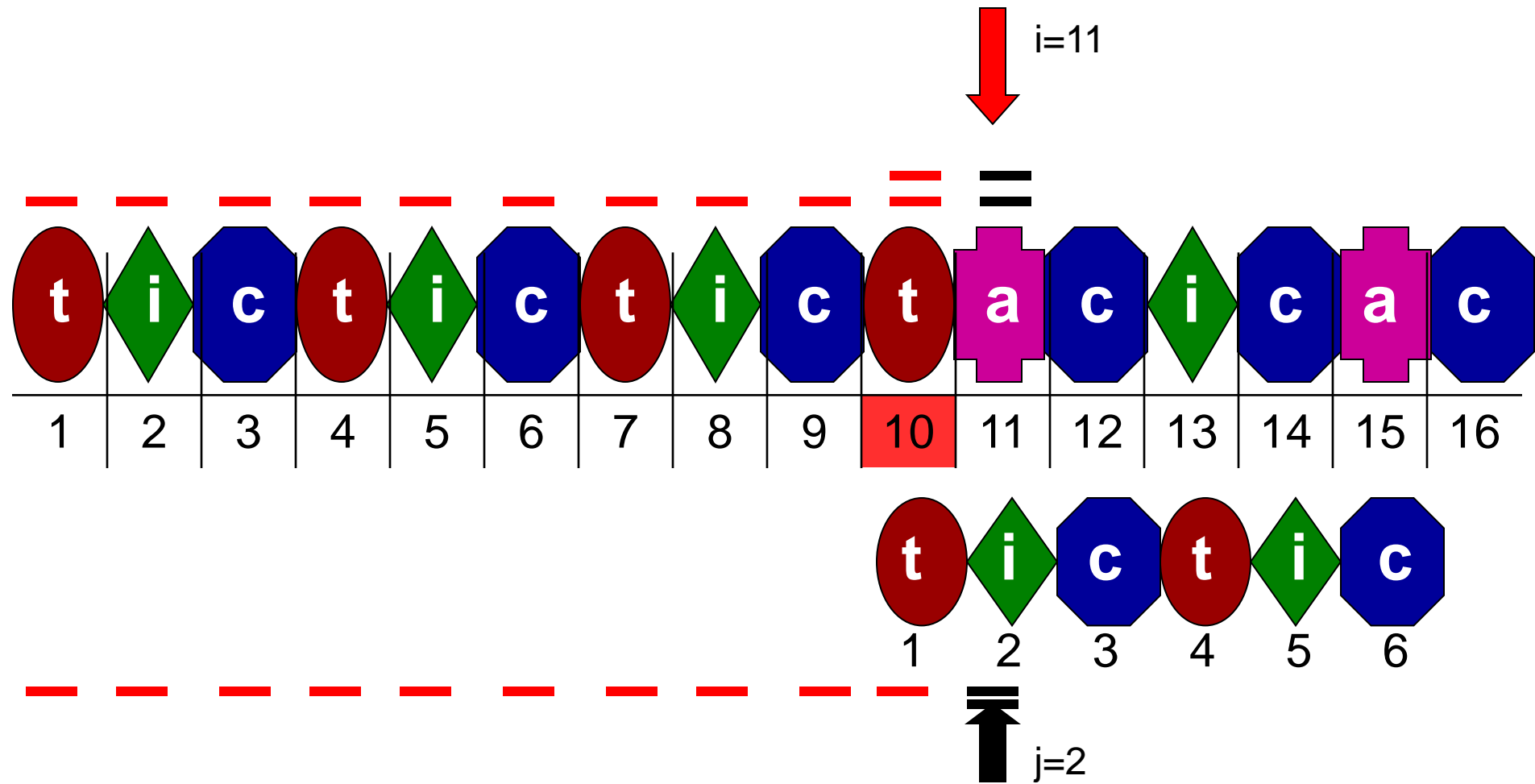
KMP search: overlap 1



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.

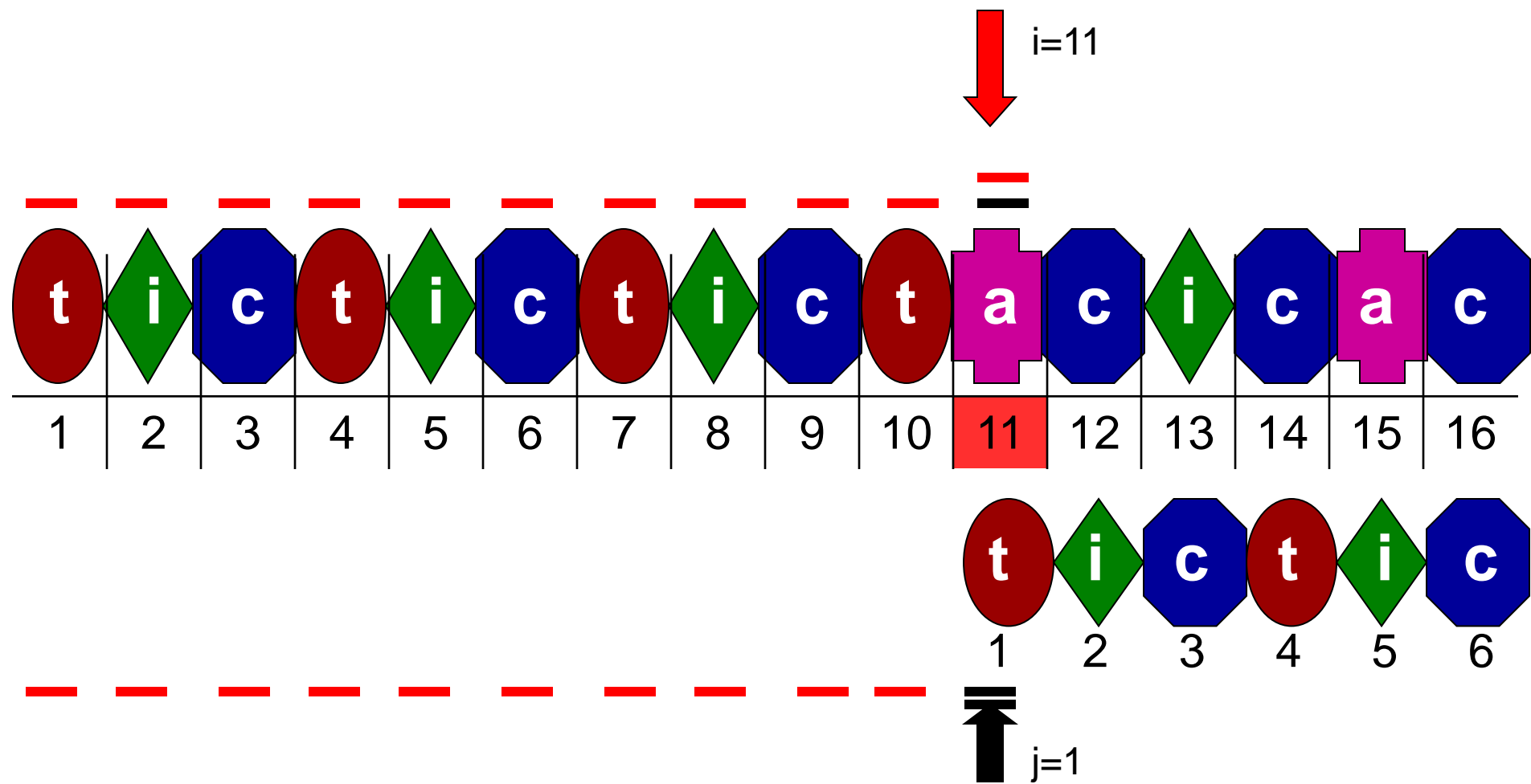
KMP search: overlap 0



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 .

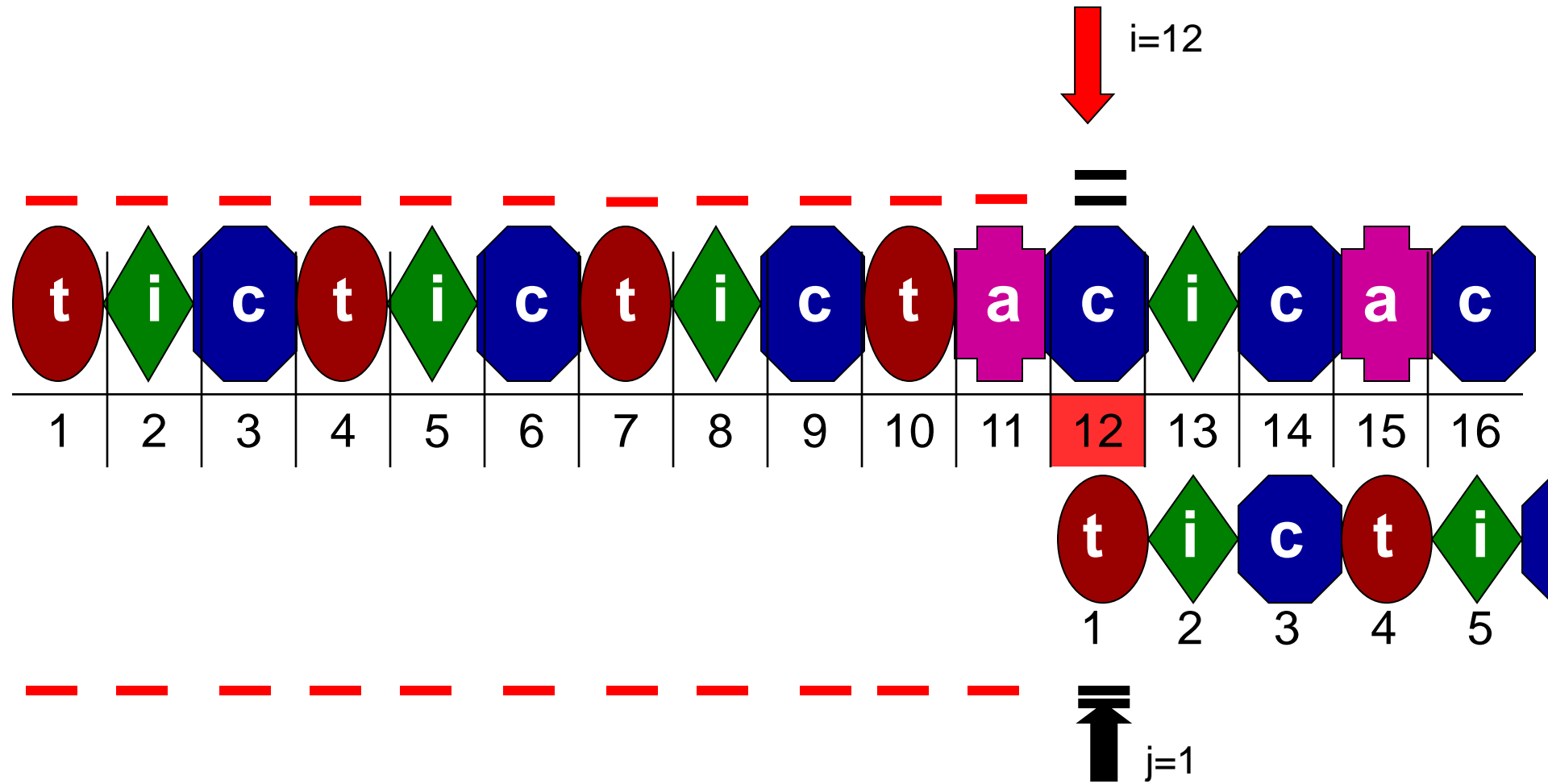
KMP search: no matches at all



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:= 'tictictictactictictic'

P:= 'tictic'

N:= len(**T**)

M:= len(**P**)

OF:= [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 *OF*(**M**) to compute pattern shift

If *j* ≠ **M** and the next characters **T**[*i*] and **P**[*j*] do 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. *OF*(*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. *OF*(*j*-1)>0. Compute pattern shift and continue comparing from the next *j*

UNTIL *i* < **N**

KMP– code

```
T:= 'tictictictactictictic'
```

```
P:= 'tictic'
```

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

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

```
OF:= [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
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 $OF(M)$ to compute pattern shift

If $j \neq M$ and the next characters $T[i]$ and $P[j]$ do not match:

See how many characters matched - 3 cases:

1. matched 0 characters: advance i , restart $j=0$

(as we would do without KMP)

2. $OF(j-1) = 0$. Previous match is not useful

so we start comparing $P[0]$ with $T[i]$
without advancing i

3. $OF(j-1) > 0$. Compute pattern shift and

continue comparing from the next j
and the same i

UNTIL $i < N$

```
 $i = 0$  # current position to compare character in T  
 $j = 0$  # current position to compare character in P
```

```
while  $i < N$ :
```

```
    # loop while characters match
```

```
    while  $j < M$  and  $i < N$  and  $T[i] == P[j]$ :
```

```
         $i = i + 1$ 
```

```
         $j = j + 1$ 
```

```
    if  $j == M$ :
```

```
        matches.append(( $i - M$ ))
```

```
    if  $j == 0$ :
```

```
         $i = i + 1$ 
```

```
    else:
```

```
         $j = of\_list[j - 1]$ 
```

```
return matches
```


KMP algorithm: time complexity

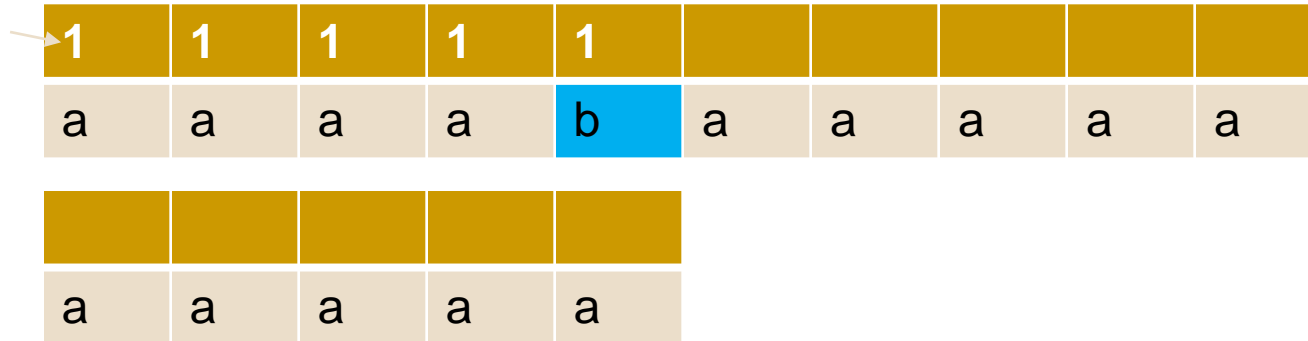
Theorem: The number of character comparisons in the KMP algorithm is at most $2N$

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 to the 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 $2N$
-

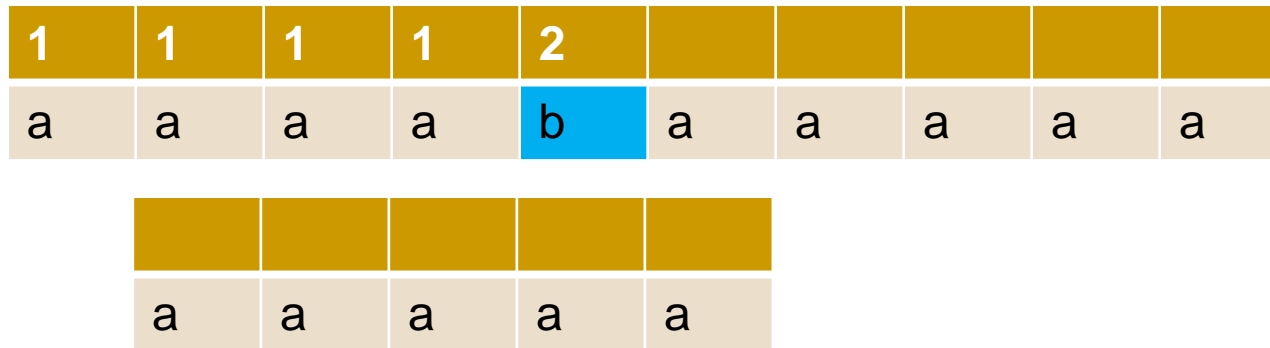
Worst-case example – iterations 1,2

Counting number
of times the
character is
accessed

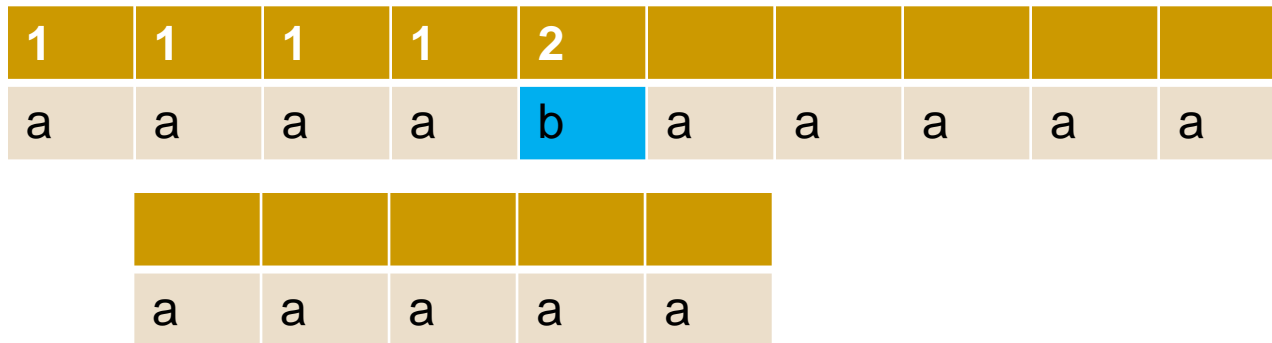


We have aligned pattern P, by performing so far 1 character comparison for each of 5 characters of P

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

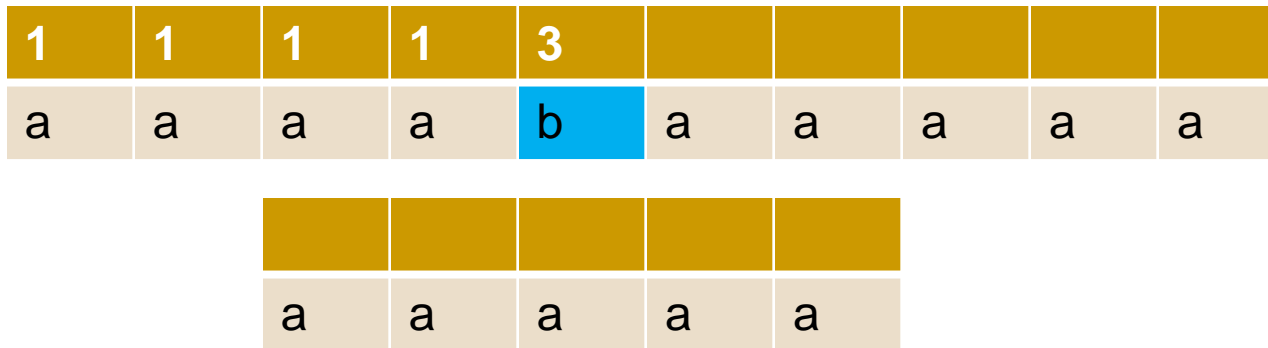


Worst-case example – iteration 3

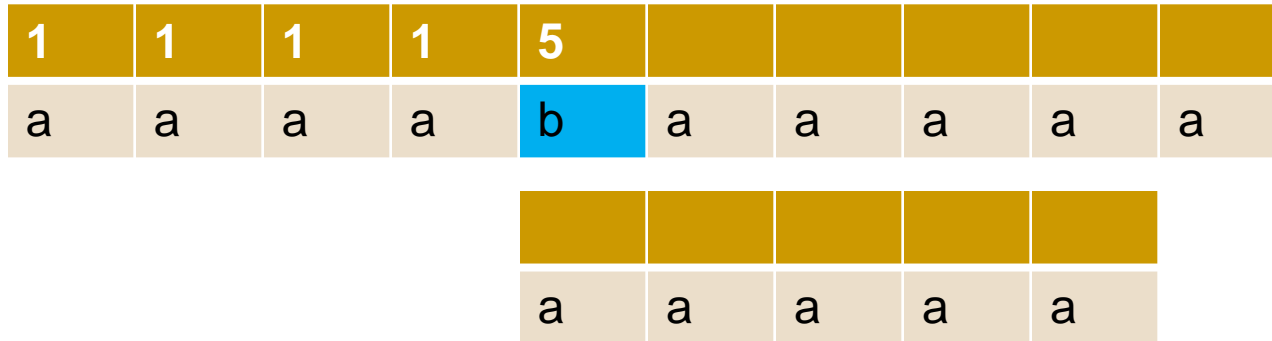


We have compared character b of T already 2 times

Next we start by aligning pattern starting at position 3 of T



Worst-case example – iteration 5



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 $2N$

Readings

- 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. [Chapter 2.3](#)
-