UNIVERSITÄT DES SAARLANDES

ZBI ZENTRUM FÜR BIOINFORMATIK

# Exact Pattern Matching: First Ideas

Algorithms for Sequence Analysis

Sven Rahmann

Summer 2021

# Pattern Matching Problem

```
alicewasbeginningtogetverytiredofsittingbyhersisteron
thebankandofhavingnothingtodoonceortwiceshehadpeepedi
ntothebookhersisterwasreadingbutithadnopicturesorconv
ersationsinitandwhatistheuseofabookthoughtalicewithou
tpicturesorconversation
```

### Task

Find all occurrences of a given string in another (longer) string.

### Goals

- As **fast** as possible (running time)
- As **easily** as possible (algorithm/implementation)

# Relevance and Applications of String Matching

## General Applications

- Web search
- Full-text searches in scientific articles
- Edit-replace in source code ...

## Applications in Computational Biology

- Searching for sequence features like binding sites
- Searching sequence data bases ("blasting")
- Building overlap graphs for de novo assembly
- Mapping next-generation sequencing reads to reference genome
- ... many many more

# Notation

| | |
|---|---|
| $\Sigma$ | alphabet = finite set of characters (letters) |
| $w \in \Sigma^k$ | string (word, $k$-gram, $k$-mer, text) of length $k$ |
| $w \in \Sigma^* = \bigcup_{k=0}^{\infty} \Sigma^k$ | word of arbitrary finite length |
| $w[i]$ | character at index $i$ in word $w$ |
| $w[i \dots j]$ | substring from $i$ to $j$ (inclusively) |

## Example

$$\Sigma = \{\mathtt{A}, \mathtt{B}, \mathtt{C}\} \qquad w[1] = \mathtt{B}$$

$$w = \mathtt{A\,B\,C\,C\,B\,A\,A\,B} \qquad w[5] = \mathtt{A}$$

$$\text{Indices: } 0\,1\,2\,3\,4\,5\,6\,7 \qquad w[1 \dots 4] = \mathtt{BCCB}$$

**Note:** Indexing starts at zero (0) !

# Pattern Matching Problem

### Given

Finite alphabet $\Sigma$, text $T \in \Sigma^n$, pattern $P \in \Sigma^m$; usually $m \ll n$.
(The pattern is a simple string for now.)

### Sought (three variants)

1. Decision: Is $P$ a substring of $T$?
   $\rightsquigarrow$ Is there an $i \in \mathbb{N}$ such that $P = T[i \dots i+m-1]$ ?
2. Counting: How often does $P$ occur in $T$?
   $\rightsquigarrow$ Let $M := \{i \in \mathbb{N} \mid P = T[i \dots i+m-1]\}$. Report $|M|$.
3. Enumeration: At what positions does $P$ occur in $T$?
   $\rightsquigarrow$ Report the full set $M$ of match positions.

# Problem Variants I

## Exact Pattern Matching (what we do next)

Given a pattern $P \in \Sigma^m$ and a text $T \in \Sigma^n$,
find indices $i$ such that $P = T[i \ldots i + m - 1]$.

# Problem Variants I

## Exact Pattern Matching (what we do next)

Given a pattern $P \in \Sigma^m$ and a text $T \in \Sigma^n$,
find indices $i$ such that $P = T[i \dots i + m - 1]$.

## Approximative Pattern Matching (later in this course)

Find all **approximate** occurrences of $P$ in $T$, i.e. for a distance measure $d$,
find indices $i, j$ such that $d(P, T[i \dots j]) \leq k$.

## Example: Hamming distance

Hamming distance: number of different positions (for strings of the same length)
$P = \texttt{ABCDE}$, $T = \texttt{XXXABDDEYYY}$
$d(P, T[3 \dots 7]) = 1$

# Problem Variants II

Pattern $P \in \Sigma^m$ and text $T \in \Sigma^n$

## Searching without index (what we do next)

- Preprocess pattern in $O(m)$
- Search text for pattern in $O(n)$
- Search for $k$ different patterns in the same text: $O\big(k(m+n)\big)$ or $O(km+n)$

## Searching with index (what we do after that)

- Preprocess text and build index data structure in $O(n)$
- Search for pattern using index in $O(m)$
- Search for $k$ different patterns in the same text: $O(n+km)$
- Index structures are useful for many tasks beyond searching

# Exact Pattern Matching using Sliding Windows

## Approach: sliding windows

- Compare pattern $P$ with window (i.e. substring) of text $T$
- Slide window across text from left to right

## Example

| | |
|---|---|
| Text: | `AACBACCABBABCA...` |
| Pattern: | `BACCAB` ← Window |

# Exact Pattern Matching using Sliding Windows

## Approach: sliding windows

- Compare pattern $P$ with window (i.e. substring) of text $T$
- Slide window across text from left to right

## Example

Text: `AACBACCABBABCA...`

Pattern: ` BACCAB`

# Exact Pattern Matching using Sliding Windows

## Approach: sliding windows

- Compare pattern $P$ with window (i.e. substring) of text $T$
- Slide window across text from left to right

## Example

Text: `AACBACCABBABCA...`

Pattern: `  BACCAB`

# Exact Pattern Matching using Sliding Windows

## Approach: sliding windows

- Compare pattern $P$ with window (i.e. substring) of text $T$
- Slide window across text from left to right

## Example

| | |
|---|---|
| Text: | `AACBACCABBABCA...` |
| Pattern: | `   BACCAB` |

# Exact Pattern Matching using Sliding Windows

## Approach: sliding windows

- Compare pattern $P$ with window (i.e. substring) of text $T$
- Slide window across text from left to right

## Example

```
Text:      AACBACCABBABCA...
Pattern:      BACCAB
```

## Naive Algorithm

- Shift window by one position in each iteration
- Compare pattern to window content from left to right

# Code: The (few) things you need to know about Python

## Pseudocode vs. Python

- (Good) Python code is as readable as pseudo code, even if you don't know Python
- Allows you (and us) to try/test algorithms immediately

# Code: The (few) things you need to know about Python

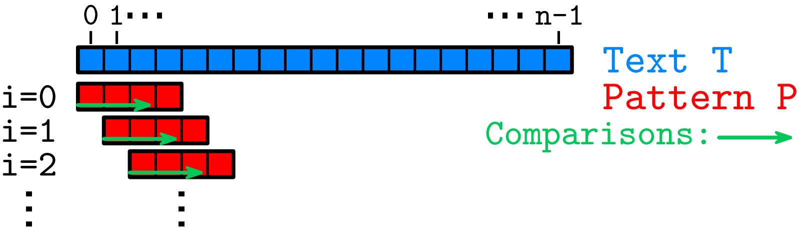
## Pseudocode vs. Python

- (Good) Python code is as readable as pseudo code, even if you don't know Python
- Allows you (and us) to try/test algorithms immediately

- "`for i in range(5,n):`": iterate over $i \in \{5, \ldots, n-1\}$
- "`for i in range(n):`": iterate over $i \in \{0, \ldots, n-1\}$
- "`len(x)`": length/size of `x`, when `x` is string, list, set, etc. (any container)
- "`T[i:j]`": substring $T[i \ldots j-1]$, also applies to lists
- "`def foo(x,y)`": **def**ine a function named `foo`
- "`yield x`": like `return`, but execution is continued later during iteration
- "`dict()`": dictionary (hash table) storing key-value pairs
- "`//`": integer division

# Naive Pattern Matching Algorithm

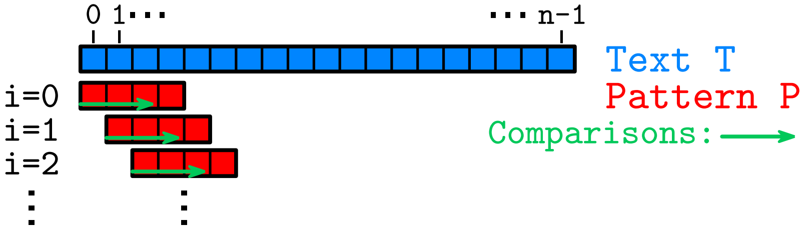
```python
def naive_pattern_matching(P, T):
  m = len(P)
  n = len(T)
  for i in range(n - m + 1):
    if T[i:i+m] == P:
      yield i
```




**Running time:** ?

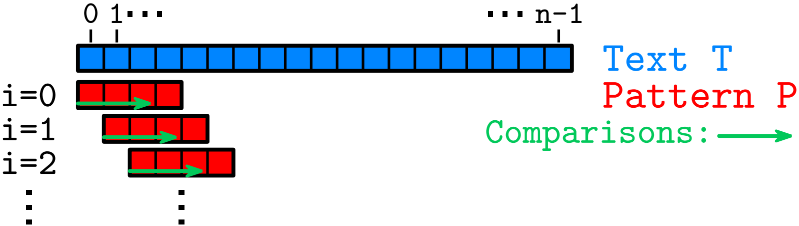
# Naive Pattern Matching Algorithm

```python
def naive_pattern_matching(P, T):
  m = len(P)
  n = len(T)
  for i in range(n - m + 1):
    if T[i:i+m] == P:
      yield i
```

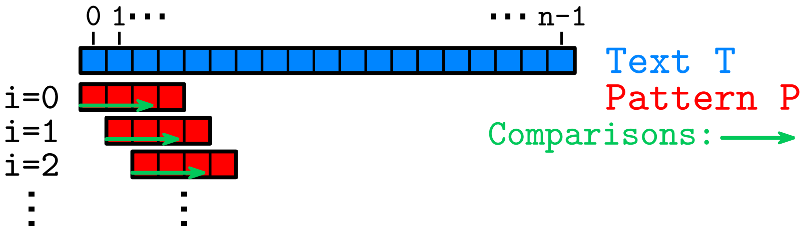



**Running time:** $O(mn)$ worst case. $O(E_m \cdot n)$ on average, but what is $E_m$?

# What can we do better?



# What can we do better?




## Ideas

1. Perhaps $O(mn)$ is pessimistic, and $O(E_m \cdot n)$ has a small constant $E_m$ ?
2. We "touch" the same characters in `T` multiple times.
   Can we "reuse" information from preceeding comparisons?
   $\rightarrow$ Automata-based algorithms (next lecture)
3. Can we shift window by more than one character?
   $\rightarrow$ Horspool algorithm (and others; next topic)

# Average-Case Analysis of the Naïve Algorithm

### Theorem (Expected Running Time)

*Let $\Sigma$ be an alphabet with $|\Sigma| \geq 2$.*
*Randomly (i.i.d.) choose a pattern of length $m$ and a text of length $n$ over $\Sigma$.*
*Then the worst-case running time of the naïve algorithnm is $O(mn)$,*
*but the expected running time is $O(E_m \cdot n) = O(n)$ with a small constant $E_m < 2$.*

# Average-Case Analysis of the Naïve Algorithm

**Theorem (Expected Running Time)**

*Let $\Sigma$ be an alphabet with $|\Sigma| \geq 2$.*
*Randomly (i.i.d.) choose a pattern of length $m$ and a text of length $n$ over $\Sigma$.*
*Then the worst-case running time of the naïve algorithnm is $O(mn)$,*
*but the expected running time is $O(E_m \cdot n) = O(n)$ with a small constant $E_m < 2$.*

We compute $E_m$: The probability $p$ that two random characters agree is

$$p := \frac{|\Sigma|}{|\Sigma|^2} = \frac{1}{|\Sigma|}.$$

(If different characters $a$ have different probabilities $p_a$ each, the expression for $p$ is more complicated, but the rest of the proof remains unchanged.)

# Average-Case Analysis of the Naïve Algorithm (Continued)

- The probability that all $m$ pattern characters match with the text window is $p^m$. This needs $m$ comparisons (and results in a match, but this is irrelevant).

# Average-Case Analysis of the Naïve Algorithm (Continued)

- The probability that all $m$ pattern characters match with the text window is $p^m$. This needs $m$ comparisons (and results in a match, but this is irrelevant).
- The probability to first fail at the $j$-th comparison $j = 1, \ldots, m$, is $p^{j-1}(1-p)$. This needs $j$ comparisons of course (and results in a mismatch).

# Average-Case Analysis of the Naïve Algorithm (Continued)

- The probability that all $m$ pattern characters match with the text window is $p^m$. This needs $m$ comparisons (and results in a match, but this is irrelevant).
- The probability to first fail at the $j$-th comparison $j = 1, \ldots, m$, is $p^{j-1}(1-p)$. This needs $j$ comparisons of course (and results in a mismatch).
- Therefore, we can compute $E_m$ as the weighted average

$$E_m := m\,p^m + \sum_{j=1}^{m} j\,p^{j-1}\,(1-p)$$

# Average-Case Analysis of the Naïve Algorithm (Continued)

- The probability that all $m$ pattern characters match with the text window is $p^m$. This needs $m$ comparisons (and results in a match, but this is irrelevant).
- The probability to first fail at the $j$-th comparison $j = 1, \ldots, m$, is $p^{j-1}(1-p)$. This needs $j$ comparisons of course (and results in a mismatch).
- Therefore, we can compute $E_m$ as the weighted average

$$E_m := m\,p^m + \sum_{j=1}^{m} j\,p^{j-1}(1-p)$$

- For any $m$, the value of $E_m$ is bounded by $E_\infty := \lim_{m\to\infty} E_m$:

$$E_m < E_\infty = (1-p)\sum_{j=1}^{\infty} j\,p^{j-1}\,.$$

# Average-Case Analysis of the Naïve Algorithm (Continued)

- The probability that all $m$ pattern characters match with the text window is $p^m$. This needs $m$ comparisons (and results in a match, but this is irrelevant).
- The probability to first fail at the $j$-th comparison $j = 1, \dots, m$, is $p^{j-1}(1-p)$. This needs $j$ comparisons of course (and results in a mismatch).
- Therefore, we can compute $E_m$ as the weighted average

$$E_m := m\, p^m + \sum_{j=1}^{m} j\, p^{j-1} (1-p)$$

- For any $m$, the value of $E_m$ is bounded by $E_\infty := \lim_{m\to\infty} E_m$:

$$E_m < E_\infty = (1-p) \sum_{j=1}^{\infty} j\, p^{j-1}\,.$$

- It remains to evaluate the series $E_\infty$ (by first-year maths).

# Average-Case Analysis of the Naïve Algorithm (Continued)

- So far: For any alphabet $\Sigma$ and any $m \geq 1$, we have

$$E_m < E_\infty \;=\; (1-p)\sum_{j=1}^{\infty} j\, p^{j-1} \;=\; (1-p)\sum_{j=\mathbf{0}}^{\infty} j\, p^{j-1}\,.$$

- You can use a computer algebra system to evaluate this, or...

# Average-Case Analysis of the Naïve Algorithm (Continued)

- So far: For any alphabet $\Sigma$ and any $m \geq 1$, we have

$$E_m < E_\infty \;=\; (1-p)\sum_{j=1}^{\infty} j\, p^{j-1} \;=\; (1-p)\sum_{j=\mathbf{0}}^{\infty} j\, p^{j-1}\,.$$

- You can use a computer algebra system to evaluate this, or...
- Consider $E_\infty = E_\infty(p)$ as a function of $p$ (recall $p = 1/|\Sigma| < 1$ for $|\Sigma| \geq 2$).
- The term $j\, p^{j-1}$ is the derivative of $p^j$.

# Average-Case Analysis of the Naïve Algorithm (Continued)

- So far: For any alphabet $\Sigma$ and any $m \geq 1$, we have

$$E_m < E_\infty \;=\; (1-p)\sum_{j=1}^{\infty} j\,p^{j-1} \;=\; (1-p)\sum_{j=\mathbf{0}}^{\infty} j\,p^{j-1}\,.$$

- You can use a computer algebra system to evaluate this, or...
- Consider $E_\infty = E_\infty(p)$ as a function of $p$ (recall $p = 1/|\Sigma| < 1$ for $|\Sigma| \geq 2$).
- The term $j\,p^{j-1}$ is the derivative of $p^j$.
- Because $\sum_{j=0}^{\infty} p^j = 1/(1-p)$ (**geometric series**), we have

$$\sum_{j=0}^{\infty} j\,p^j = \frac{\mathrm{d}}{\mathrm{d}p}\frac{1}{1-p} = \frac{1}{(1-p)^2}\,,$$

$$E_\infty = \frac{1-p}{(1-p)^2} = \frac{1}{1-p} = \frac{|\Sigma|}{|\Sigma|-1} \leq 2.$$

# Average-Case Analysis of the Naïve Algorithm (Conclusion)

- In summary, for all $m \geq 1$ and all $\Sigma \geq 2$,

$$E_m < E_\infty = \frac{1}{1-p} = \frac{|\Sigma|}{|\Sigma|-1} \leq 2\,.$$

- For $|\Sigma| \to \infty$ we have $E_m \searrow 1$.
- The expected running time on i.i.d. random texts is thus $O(n \cdot E_m) = O(n)$ with a small constant $E_m \leq 2$.

# Horspool Algorithm

# Motivation: Horspool Algorithm

## Question

When and how can the window be shifted by more than one position?

## Ideas

- Compare pattern **right-to-left** to text window.
- Characters not occurring in pattern $\rightarrow$ large shift

## (Extreme) Example

```
AAAAAAAAAAAAAAAAAAAAAAAAAA
BBBBBB
```

# Motivation: Horspool Algorithm

## Question

When and how can the window be shifted by more than one position?

## Ideas

- Compare pattern **right-to-left** to text window.
- Characters not occurring in pattern $\rightarrow$ large shift

## (Extreme) Example

```
AAAAAAAAAAAAAAAAAAAAAAAAAAA
      BBBBBB
```

# Motivation: Horspool Algorithm

## Question

When and how can the window be shifted by more than one position?

## Ideas

- Compare pattern **right-to-left** to text window.
- Characters not occurring in pattern → large shift

## (Extreme) Example

```
AAAAAAAAAAAAAAAAAAAAAAAAAAA
            BBBBBB
```

# Motivation: Horspool Algorithm

## Question

When and how can the window be shifted by more than one position?

## Ideas

- Compare pattern **right-to-left** to text window.
- Characters not occurring in pattern $\rightarrow$ large shift

## (Extreme) Example

```
AAAAAAAAAAAAAAAAAAAAAAAAAAA
                  BBBBBB
```

# Motivation: Horspool Algorithm

## Question

When and how can the window be shifted by more than one position?

## Ideas

- Compare pattern **right-to-left** to text window.
- Characters not occurring in pattern $\rightarrow$ large shift

## (Extreme) Example

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAA
                     BBBBBB
```

**Best case** time: $O(n/m)$

# Horspool Algorithm

## Approach

- Window-based pattern matching algorithm
- Shift determined by **last character** in window

## $P = \mathtt{BAAAAB}$ and $\Sigma = \{\mathtt{A}, \mathtt{B}, \mathtt{C}\}$

**Question:** How far can we shift the window without missing pattern occurrences?

Text: $\underbrace{\mathtt{?????A}}\mathtt{?????}$ $\underbrace{\mathtt{?????B}}\mathtt{?????}$ $\underbrace{\mathtt{?????C}}\mathtt{??????}$

# Horspool Algorithm

## Approach

- Window-based pattern matching algorithm
- Shift determined by **last character** in window

## $P = \texttt{BAAAAB}$ and $\Sigma = \{\texttt{A}, \texttt{B}, \texttt{C}\}$

**Question:** How far can we shift the window without missing pattern occurrences?

Text: `?????A?????` `?????B?????` `?????C??????`

Pattern: `BAAAAB` `BAAAAB` `BAAAAB`

# Horspool Algorithm (formal) I

$p = \texttt{BAAAAB}$ und $\Sigma = \{\texttt{A}, \texttt{B}, \texttt{C}\}$

| | | | |
|---|---|---|---|
| Text: | `?????A?????` | `?????B?????` | `?????C??????` |
| Pattern: | `BAAAAB` | `BAAAAB` | `BAAAAB` |

```python
def horspool_preprocessing(sigma, P):
  shifts = dict()
  for c in sigma:
    shifts[c] = len(P)
  for i in range(len(P)-1):
    shifts[P[i]] = len(P) - i - 1
  return shifts
```

# Horspool Algorithm (formal) II

```
def horspool_preprocessing(sigma, P):
    shifts = dict()
    for c in sigma:
        shifts[c] = len(P)
    for i in range(len(P)-1):
        shifts[P[i]] = len(P) - i - 1
    return shifts
```

# Horspool Algorithm (formal) II

```
def horspool_preprocessing(sigma, P):
    shifts = dict()
    for c in sigma:
        shifts[c] = len(P)
    for i in range(len(P)-1):
        shifts[P[i]] = len(P) - i - 1
    return shifts
```

shifts:

| A | B | C |
|---|---|---|
| 6 | 6 | 6 |

# Horspool Algorithm (formal) II

```python
def horspool_preprocessing(sigma, P):
    shifts = dict()
    for c in sigma:
        shifts[c] = len(P)
    for i in range(len(P)-1):
        shifts[P[i]] = len(P) - i - 1
    return shifts
```

shifts:

| A | B | C |
|---|---|---|
| 6 | 5 | 6 |

$i = 0 \qquad p = \texttt{BAAAAB}$

# Horspool Algorithm (formal) II

```python
def horspool_preprocessing(sigma, P):
    shifts = dict()
    for c in sigma:
        shifts[c] = len(P)
    for i in range(len(P)-1):
        shifts[P[i]] = len(P) - i - 1
    return shifts
```

shifts:

| A | B | C |
|---|---|---|
| 4 | 5 | 6 |

$i = 1$ $p =$ BAAAAB

# Horspool Algorithm (formal) II

```
def horspool_preprocessing(sigma, P):
    shifts = dict()
    for c in sigma:
        shifts[c] = len(P)
    for i in range(len(P)-1):
        shifts[P[i]] = len(P) - i - 1
    return shifts
```

shifts:

| A | B | C |
|---|---|---|
| 3 | 5 | 6 |

$i = 2$ $p =$ BAAAAB

# Horspool Algorithm (formal) II

```
def horspool_preprocessing(sigma, P):
    shifts = dict()
    for c in sigma:
        shifts[c] = len(P)
    for i in range(len(P)-1):
        shifts[P[i]] = len(P) - i - 1
    return shifts
```

shifts:

| A | B | C |
|---|---|---|
| 2 | 5 | 6 |

$i = 3$ $p =$ BAAAAB

# Horspool Algorithm (formal) II

```
def horspool_preprocessing(sigma, P):
    shifts = dict()
    for c in sigma:
        shifts[c] = len(P)
    for i in range(len(P)-1):
        shifts[P[i]] = len(P) - i - 1
    return shifts
```

shifts:

| A | B | C |
|---|---|---|
| 1 | 5 | 6 |

$i = 4$ $p =$ BAAAAB

# Horspool-Algorithmus (formal) III

```python
def horspool_matching(sigma, P, T):
  shifts =  horspool_preprocessing(sigma, P)
  i = len(P) - 1
  while i < len(T):
    if T[i-len(P)+1:i+1] == P:
      yield i
    i += shifts[T[i]]
```

# Horspool-Algorithmus (formal) III

```python
def horspool_matching(sigma, P, T):
  shifts =  horspool_preprocessing(sigma, P)
  i = len(P) - 1
  while i < len(T):
    if T[i-len(P)+1:i+1] == P:
      yield i
    i += shifts[T[i]]
```

Text: ABBCACBABAABBAAAABAABCAC...

BAAAAB

shifts:

| A | B | C |
|---|---|---|
| 1 | 5 | 6 |

# Horspool-Algorithmus (formal) III

```
def horspool_matching(sigma, P, T):
  shifts =  horspool_preprocessing(sigma, P)
  i = len(P) - 1
  while i < len(T):
    if T[i-len(P)+1:i+1] == P:
      yield i
    i += shifts[T[i]]
```

Text: ABBCACBABAABBAAAABAABCAC...

BAAAAB

shifts:

| A | B | C |
|---|---|---|
| 1 | 5 | 6 |

# Horspool-Algorithmus (formal) III

```python
def horspool_matching(sigma, P, T):
  shifts = horspool_preprocessing(sigma, P)
  i = len(P) - 1
  while i < len(T):
    if T[i-len(P)+1:i+1] == P:
      yield i
    i += shifts[T[i]]
```

Text: ABBCACBABAABBAAAABAABCAC...

BAAAAB

shifts:

| A | B | C |
|---|---|---|
| 1 | 5 | 6 |

# Horspool-Algorithmus (formal) III

```python
def horspool_matching(sigma, P, T):
    shifts =  horspool_preprocessing(sigma, P)
    i = len(P) - 1
    while i < len(T):
        if T[i-len(P)+1:i+1] == P:
            yield i
        i += shifts[T[i]]
```

Text: ABBCACBABAABBAAAABAABCAC...

BAAAAB

shifts:

| A | B | C |
|---|---|---|
| 1 | 5 | 6 |

# Horspool-Algorithmus (formal) III

```
def horspool_matching(sigma, P, T):
  shifts = horspool_preprocessing(sigma, P)
  i = len(P) - 1
  while i < len(T):
    if T[i-len(P)+1:i+1] == P:
      yield i
    i += shifts[T[i]]
```

Text: ABBCACBABAAB**BAAAAB**AABCAC...

$\underbrace{\texttt{BAAAAB}}$

BAAAAB

shifts:

| A | B | C |
|---|---|---|
| 1 | 5 | 6 |

### Property of the Horspool Algorithm

- Fast for large alphabets and long patterns

# Summary

- Today's topic: **Exact Pattern Matching** (for single patterns without index)
- Naïve algorithm and analysis
- Idea to improve on naive algorithm:
  Shift window by more than one character
  $\rightarrow$ **Horspool algorithm**

# Possible exam questions

- State the pattern matching problem and known variants of it.
- What is the worst-case and average-case running time of the naïve algorithm?
- The naïve algorithm is fast on average; why bother with more complex algorithms?
- Explain Horspool's algorithm.
- Construct the Horspool shift table for a short pattern.
- For which pattern properties is Horspool's algorithm fast or slow?
- How may Horspool's algorithm be modified to be fast on long patterns with small alphabet?