Basic Definitions and Exact Pattern Search

Algorithms for Sequence Analysis

Sven Rahmann

Summer 2024

Introduction

The Exact Pattern Search Problem (Exact Matching)

```
alicewasbeginningtogetverytiredofsittingbyhersisteron
thebankandofhavingnothingtodoonceortwiceshehadpeepedi
ntothebookhersisterwasreadingbutithadnopicturesorconv
ersationsinitandwhatistheuseofabookthoughtalicewithou
tpicturesorconversation
```

The Exact Pattern Search Problem (Exact Matching)

```
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

General Applications

- Web search
- Full-text searches in scientific articles
- Find + replace in source code, etc.

Relevance and Applications

General Applications

- Web search
- Full-text searches in scientific articles
- Find + replace in source code, etc.

Applications in Computational Biology

- Searching for DNA/protein sequence features like binding sites
- Searching sequence data bases ("blasting")
- Building overlap graphs for de novo assembly
- Mapping sequenced DNA reads to a reference genome

Basic Definitions

Basic Notation

Σ	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
$w[i]$ or w_i	character at index i in word w (indexing starts at **zero**!)
$w[i \ldots j]$	substring from i to j (inclusively)
`w[i:j]`	substring from i to $j - 1$ (excluding j), Python notation

Basic Notation

Σ	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
$w[i]$ or w_i	character at index i in word w (indexing starts at **zero**!)
$w[i \dots j]$	substring from i to j (inclusively)
`w[i:j]`	substring from i to $j-1$ (excluding j), Python notation

Examples

$\Sigma = \{\mathtt{A}, \mathtt{B}, \mathtt{C}\}$	$w[1] = w_1 = \mathtt{B}$
$w = \mathtt{A\,B\,C\,C\,B\,A\,A\,B}$	$w[5] = w_5 = \mathtt{A}$
Indices: $0\,1\,2\,3\,4\,5\,6\,7$	$w[1 \dots 4] = w[1:5] = \mathtt{BCCB}$

Definitions

Special substrings

Let w be a string of length n (written $|w| = n$).

prefix	any $w[0:i] = w[:i]$, any substring that starts at index 0
suffix	any $w[i:n] = w[i:]$, any substring that ends at the last character w_{n-1}
empty	string ε, the(!) string of length zero (over any alphabet)
proper	any prefix, suffix that is not empty and not equal to w
substring	any $w[i:j]$ with $0 \leq i \leq j \leq n$; w_j not inlcuded; length is $j - i$
subsequence	$(w_i)_{i \in I}$, where I is a subset of $\{0, 1, \ldots, n-1\}$
k-mer	any substring of the given length k

Definitions

Special substrings

Let w be a string of length n (written $|w| = n$).

prefix	any $w[0:i] = w[:i]$, any substring that starts at index 0
suffix	any $w[i:n] = w[i:]$, any substring that ends at the last character w_{n-1}
empty	string ε, the(!) string of length zero (over any alphabet)
proper	any prefix, suffix that is not empty and not equal to w
substring	any $w[i:j]$ with $0 \leq i \leq j \leq n$; w_j not inlcuded; length is $j - i$
subsequence	$(w_i)_{i \in I}$, where I is a subset of $\{0, 1, \ldots, n-1\}$
k-mer	any substring of the given length k

Example

Let $w = \texttt{GATTACA}$.

`ATTAC` is a substring ($w[1:6]$, a 5-mer), but not a prefix or suffix.

`AAA` is not a substring, but a subsequence (index set $I = \{1, 4, 6\}$).

Integer Encoding of Substrings (k-mers)

Translation of characters to small integers

Let Σ be an alphabet with $|\Sigma| = s$.
Then we can find a bijection ("translation") $t : \Sigma \to \{0, 1, \dots, s-1\}$.
Example for DNA with $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$: $\mathtt{A} \mapsto 0$, $\mathtt{C} \mapsto 1$, $\mathtt{G} \mapsto 2$, $\mathtt{T} \mapsto 3$.
Different bijections ($s!$ in fact) are possible; usually there is a "natural" one.

Integer Encoding of Substrings (k-mers)

Translation of characters to small integers

Let Σ be an alphabet with $|\Sigma| = s$.
Then we can find a bijection ("translation") $t : \Sigma \to \{0, 1, \dots, s-1\}$.
Example for DNA with $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$: $\mathtt{A} \mapsto 0$, $\mathtt{C} \mapsto 1$, $\mathtt{G} \mapsto 2$, $\mathtt{T} \mapsto 3$.
Different bijections ($s!$ in fact) are possible; usually there is a "natural" one.

Encoding k-mers

For fixed (small) k, we may encode k-mers as (small) integers in $\{0, \dots, s^k - 1\}$, by reading the translated k-mer as a base-s number with k digits.

Integer Encoding of Substrings (k-mers)

Translation of characters to small integers

Let Σ be an alphabet with $|\Sigma| = s$.
Then we can find a bijection ("translation") $t : \Sigma \to \{0, 1, \dots, s-1\}$.
Example for DNA with $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$: $\mathtt{A} \mapsto 0$, $\mathtt{C} \mapsto 1$, $\mathtt{G} \mapsto 2$, $\mathtt{T} \mapsto 3$.
Different bijections ($s!$ in fact) are possible; usually there is a "natural" one.

Encoding k-mers

For fixed (small) k, we may encode k-mers as (small) integers in $\{0, \dots, s^k - 1\}$,
by reading the translated k-mer as a base-s number with k digits.
Example for DNA with $k = 3$: $\mathtt{CTG} \mapsto (132)_4 = 1 \cdot 16 + 3 \cdot 4 + 2 \cdot 1 = 30$.

Integer Encoding of Substrings (k-mers)

Translation of characters to small integers

Let Σ be an alphabet with $|\Sigma| = s$.
Then we can find a bijection ("translation") $t : \Sigma \to \{0, 1, \ldots, s-1\}$.
Example for DNA with $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$: $\mathtt{A} \mapsto 0$, $\mathtt{C} \mapsto 1$, $\mathtt{G} \mapsto 2$, $\mathtt{T} \mapsto 3$.
Different bijections ($s!$ in fact) are possible; usually there is a "natural" one.

Encoding k-mers

For fixed (small) k, we may encode k-mers as (small) integers in $\{0, \ldots, s^k - 1\}$,
by reading the translated k-mer as a base-s number with k digits.
Example for DNA with $k = 3$: $\mathtt{CTG} \mapsto (132)_4 = 1 \cdot 16 + 3 \cdot 4 + 2 \cdot 1 = 30$.
Example for DNA with $k = 5$: $\mathtt{AACTG} \mapsto 30$ as well; so k needs to be fixed.

Canonical codes for DNA k-mers

What is special about DNA strings?

DNA is double stranded.
A DNA word w is **equivalent** to its reverse complement ($\mathtt{A} \leftrightarrow \mathtt{T}$, $\mathtt{C} \leftrightarrow \mathtt{G}$) $revcomp(w)$.
Therefore, w and $revcomp(w)$ should get the same encoding (**canonical code**):
achieved by using the **smaller** encoding of w and $revcomp(w)$ for both of them.

Canonical codes for DNA k-mers

What is special about DNA strings?

DNA is double stranded.
A DNA word w is **equivalent** to its reverse complement ($\mathtt{A} \leftrightarrow \mathtt{T}$, $\mathtt{C} \leftrightarrow \mathtt{G}$) $\mathit{revcomp}(w)$.
Therefore, w and $\mathit{revcomp}(w)$ should get the same encoding (**canonical code**):
achieved by using the **smaller** encoding of w and $\mathit{revcomp}(w)$ for both of them.

Example for $k = 3$

The reverse complement of `CTG` is `CAG`.
$\mathtt{CTG} \mapsto (132)_4 = 1 \cdot 16 + 3 \cdot 4 + 2 \cdot 1 = 30.$
$\mathtt{CAG} \mapsto (102)_4 = 1 \cdot 16 + 0 \cdot 4 + 2 \cdot 1 = 18.$
The smaller value is 18, so the canonical code for both `CTG` and `CAG` is 18.

Canonical codes for DNA k-mers

What is special about DNA strings?

DNA is double stranded.

A DNA word w is **equivalent** to its reverse complement ($\mathtt{A} \leftrightarrow \mathtt{T}$, $\mathtt{C} \leftrightarrow \mathtt{G}$) $\mathit{revcomp}(w)$.

Therefore, w and $\mathit{revcomp}(w)$ should get the same encoding (**canonical code**):

achieved by using the **smaller** encoding of w and $\mathit{revcomp}(w)$ for both of them.

Example for $k = 3$

The reverse complement of `CTG` is `CAG`.

$\mathtt{CTG} \mapsto (132)_4 = 1 \cdot 16 + 3 \cdot 4 + 2 \cdot 1 = 30.$

$\mathtt{CAG} \mapsto (102)_4 = 1 \cdot 16 + 0 \cdot 4 + 2 \cdot 1 = 18.$

The smaller value is 18, so the canonical code for both `CTG` and `CAG` is 18.

Canonical codes are important in practice; make sure you understand them.

Exact Pattern Search

Exact Pattern Search (Matching) Problem

Given

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

Exact Pattern Search (Matching) Problem

Given

finite alphabet Σ, 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 $M := \{ i \in \mathbb{N} \,|\, P = T[i \dots i+m-1] \}$. 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 Search (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 Search (what we do next)

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

Approximate Pattern Search (later)

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

Example: Hamming distance

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

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: $O(k(m+n))$ 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 Search 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

Naive Algorithm

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

Example

Text: `AACBACCABBABCA...`

Pattern: `BACCAB` ← Window

Exact Pattern Search 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

Naive Algorithm

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

Example

Text: `AACBACCABBABCA...`

Pattern: ` BACCAB`

Exact Pattern Search 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

Naive Algorithm

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

Example

Text: `AACBACCABBABCA...`

Pattern: `  BACCAB`

Exact Pattern Search 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

Naive Algorithm

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

Example

Text: `AACBACCABBABCA...`

Pattern: `BACCAB`

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

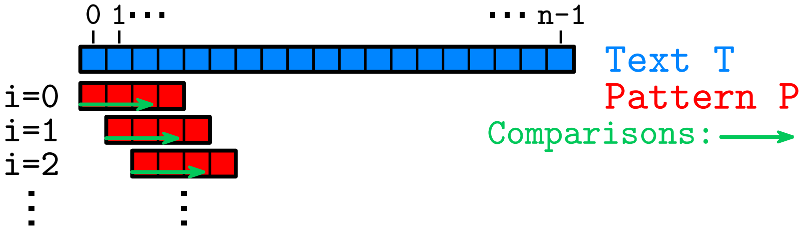
Pseudocode vs. Python

- (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

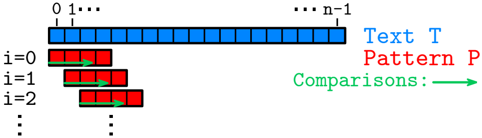
- `for i in range(5,n):` iterate over $i \in \{5, \dots, n-1\}$
- `for i in range(n):` iterate over $i \in \{0, \dots, n-1\}$
- `len(x)` length/size of `x`, when `x` is a string, list, set, any container
- `T[i:j]` substring $T[i \dots j-1]$, also applies to lists
- `def foo(x,y):` **def**ine a function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

Naive Pattern Search Algorithm

```
def naive_pattern_search(P, T):
    m, n = len(P), len(T)
    for i in range(n - m + 1):
        if T[i:i+m] == P:   # implicit loop of size m
            yield i
```



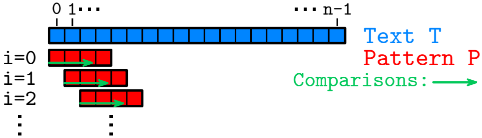
Running Time and Possible Improvements



Running time

- $O(mn)$ worst case
- $O(E_m \cdot n)$ on average; E_m: average number of comparisons per window

Running Time and Possible Improvements



Running time

- $O(mn)$ worst case
- $O(E_m \cdot n)$ on average; E_m: average number of comparisons per window

Thoughts

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 "re-use" information from preceding comparisons?
   $\rightarrow$ Automata-based algorithms
3. Can we shift window by more than one character? $\rightarrow$ Horspool algorithm

Average-Case Analysis of the Naïve Algorithm

Theorem: Expected Running Time

Let Σ be an alphabet with $|\Sigma| \geq 2$.
Randomly (i.i.d.) choose a pattern of length m and a text of length n over Σ.
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 Σ be an alphabet with $|\Sigma| \geq 2$.
Randomly (i.i.d.) choose a pattern of length m and a text of length n over Σ.
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 changes to $p = \sum_{a \in \Sigma} p_a^2$, 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 $j = 1, \ldots, m$, 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 $E_\infty := \lim_{m\to\infty} E_m$:

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

- It remains to evaluate the series E_∞.

Average-Case Analysis of the Naïve Algorithm (Continued)

- So far: For any alphabet Σ 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 Σ 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. We may swap derivative and sum.

Average-Case Analysis of the Naïve Algorithm (Continued)

- So far: For any alphabet Σ 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. We may swap derivative and sum.
- Because $\sum_{j=0}^{\infty} p^j = 1/(1-p)$ (**geometric** series), we have

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

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

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 $E_m \leq 2$.

Summary

- Basic definitions
- k-mers and integer encodings of k-mers
- Special considerations about DNA sequences (double-stranded): **canonical codes** for DNA k-mers
- **Exact Pattern Search** (for single patterns without index)
- Naïve algorithm
- Worst-case analysis: $O(mn)$ with $|P| = m$ and $|T| = n$
- Average-case analysis on random texts and patterns: $O(E_m n)$ with $E_m \leq 2$

Possible exam questions

- What is the difference between string, sequence, word, k-mer, q-gram?
- What is the difference between substring and subsequence?
- What is a k-mer ?
- How can we encode a k-mer as an integer ?
- What is special about k-mers of DNA sequences ?
- How can canonical codes of DNA k-mers be defined ?
- State the exact pattern search 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?