Algorithms and datastructures II

Lecture 1: text searching (1/2)

Jan Hubička

Department of Applied Mathematics
Charles University
Prague

Oct 5 2020

Algorithms and data-structures II

Lecturer: Jan Hubička, `hubicka@kam.mff.cuni.cz`.

- Consultations by appointment (email or zoom).
- Do we want a discussion forum, like discord or moodle?

Algorithms and data-structures II

Lecturer: Jan Hubička, `hubicka@kam.mff.cuni.cz`.

- Consultations by appointment (email or zoom).
- Do we want a discussion forum, like discord or moodle?

Lecture: Monday 9am over zoom.

- Lectures will be live: please come and do ask questions, tell me how you like the lecture!
- Recordings will be available on `stream.cuni.cz` (linked from webpage).
- Recordings will include all questions and communication. If you ask me after the lecture ends, I will edit out your question.
- I will try to keep lecture synchronized with Czech lecture by Jan Hric.
- If you spot mistake, please let me know. The slides are brand new.

Webpage: `https://iuuk.mff.cuni.cz/~hubicka/2020/adsII.html`

Syllabus

1. String-searching

Syllabus

1. String-searching
2. Network flows

Syllabus

1. String-searching
2. Network flows
3. Algebraic algorithms (Fourier transformation)

Syllabus

1. String-searching
2. Network flows
3. Algebraic algorithms (Fourier transformation)
4. Parallel algorithms

Syllabus

1. String-searching
2. Network flows
3. Algebraic algorithms (Fourier transformation)
4. Parallel algorithms
5. Geometric algorithms

Syllabus

1. String-searching
2. Network flows
3. Algebraic algorithms (Fourier transformation)
4. Parallel algorithms
5. Geometric algorithms
6. Introduction to complexity

Syllabus

1. String-searching
2. Network flows
3. Algebraic algorithms (Fourier transformation)
4. Parallel algorithms
5. Geometric algorithms
6. Introduction to complexity
7. Approximation algorithm

Syllabus

1. String-searching
2. Network flows
3. Algebraic algorithms (Fourier transformation)
4. Parallel algorithms
5. Geometric algorithms
6. Introduction to complexity
7. Approximation algorithm
8. Probabilistic algorithms and cryptography

Today: string-searching (“searching needle in a haystack”)

string-searching

Given a string ν (“a needle”) and η (“a haystack”) find all occurrences of ν in η.

Today: string-searching ("searching needle in a haystack")

string-searching

Given a string ν ("a needle") and η ("a haystack") find all occurrences of ν in η.

Some notation:

1. Σ: an alphabet (finite set of characters)

Today: string-searching (“searching needle in a haystack”)

string-searching

Given a string ν (“a needle”) and η (“a haystack”) find all occurrences of ν in η.

Some notation:

1. Σ: an alphabet (finite set of characters)
2. Σ^*: the set all finite words in alphabet Σ

Today: string-searching ("searching needle in a haystack")

string-searching

Given a string ν ("a needle") and η ("a haystack") find all occurrences of ν in η.

Some notation:

1. Σ: an alphabet (finite set of characters)
2. Σ^*: the set all finite words in alphabet Σ
3. α, β, . . . : words

Today: string-searching (“searching needle in a haystack”)

string-searching

Given a string ν (“a needle”) and η (“a haystack”) find all occurrences of ν in η.

Some notation:

1. Σ: an alphabet (finite set of characters)
2. Σ^*: the set all finite words in alphabet Σ
3. α, β, . . . : words
4. $|\alpha|$: length of the word α.

Today: string-searching (“searching needle in a haystack”)

string-searching

Given a string ν (“a needle”) and η (“a haystack”) find all occurrences of ν in η.

Some notation:

1. Σ: an alphabet (finite set of characters)
2. Σ^*: the set all finite words in alphabet Σ
3. α, β, $\ldots$: words
4. $|\alpha|$: length of the word α.
5. ϵ: empty word (the only word of length 0)

Today: string-searching (“searching needle in a haystack”)

string-searching

Given a string ν (“a needle”) and η (“a haystack”) find all occurrences of ν in η.

Some notation:

1. Σ: an alphabet (finite set of characters)
2. Σ^*: the set all finite words in alphabet Σ
3. α, β, $\ldots$: words
4. $|\alpha|$: length of the word α.
5. ϵ: empty word (the only word of length 0)
6. $\alpha\beta$: concatenation of α and β

Today: string-searching (“searching needle in a haystack”)

string-searching

Given a string ν (“a needle”) and η (“a haystack”) find all occurrences of ν in η.

Some notation:

1. Σ: an alphabet (finite set of characters)
2. Σ^*: the set all finite words in alphabet Σ
3. α, β, ...: words
4. $|\alpha|$: length of the word α.
5. ϵ: empty word (the only word of length 0)
6. $\alpha\beta$: concatenation of α and β
7. $\alpha[i]$: i-th character of α (starting from 0)

Today: string-searching (“searching needle in a haystack”)

string-searching

Given a string ν (“a needle”) and η (“a haystack”) find all occurrences of ν in η.

Some notation:

1. Σ: an alphabet (finite set of characters)
2. Σ^*: the set all finite words in alphabet Σ
3. α, β, . . . : words
4. $|\alpha|$: length of the word α.
5. ϵ: empty word (the only word of length 0)
6. $\alpha\beta$: concatenation of α and β
7. $\alpha[i]$: i-th character of α (starting from 0)
8. $\alpha[i : j]$: subword $\alpha[i]\alpha[i+1]\cdots\alpha[j-1]$

Today: string-searching (“searching needle in a haystack”)

string-searching

Given a string ν (“a needle”) and η (“a haystack”) find all occurrences of ν in η.

Some notation:

1. Σ: an alphabet (finite set of characters)
2. Σ^*: the set all finite words in alphabet Σ
3. α, β, . . . : words
4. $|\alpha|$: length of the word α.
5. ϵ: empty word (the only word of length 0)
6. $\alpha\beta$: concatenation of α and β
7. $\alpha[i]$: i-th character of α (starting from 0)
8. $\alpha[i : j]$: subword $\alpha[i]\alpha[i+1]\cdots\alpha[j-1]$
9. $\alpha[: j]$: prefix of α of length j

Today: string-searching (“searching needle in a haystack”)

string-searching

Given a string ν (“a needle”) and η (“a haystack”) find all occurrences of ν in η.

Some notation:

1. Σ: an alphabet (finite set of characters)
2. Σ^*: the set all finite words in alphabet Σ
3. α, β, . . . : words
4. $|\alpha|$: length of the word α.
5. ϵ: empty word (the only word of length 0)
6. $\alpha\beta$: concatenation of α and β
7. $\alpha[i]$: i-th character of α (starting from 0)
8. $\alpha[i : j]$: subword $\alpha[i]\alpha[i + 1] \cdots \alpha[j - 1]$
9. $\alpha[: j]$: prefix of α of length j
10. $\alpha[i :]$: a suffix of α

Today: string-searching (“searching needle in a haystack”)

string-searching

Given a string ν (“a needle”) and η (“a haystack”) find all occurrences of ν in η.

Some notation:

1. Σ: an alphabet (finite set of characters)
2. Σ^*: the set all finite words in alphabet Σ
3. α, β, ...: words
4. $|\alpha|$: length of the word α.
5. ϵ: empty word (the only word of length 0)
6. $\alpha\beta$: concatenation of α and β
7. $\alpha[i]$: i-th character of α (starting from 0)
8. $\alpha[i:j]$: subword $\alpha[i]\alpha[i+1]\cdots\alpha[j-1]$
9. $\alpha[:j]$: prefix of α of length j
10. $\alpha[i:]$: a suffix of α
11. $\alpha[:]$: whole word α

Today: string-searching (“searching needle in a haystack”)

string-searching

Given a string ν (“a needle”) and η (“a haystack”) find all occurrences of ν in η.

Some notation:

1. Σ: an alphabet (finite set of characters)
2. Σ^*: the set all finite words in alphabet Σ
3. α, β, . . . : words
4. $|\alpha|$: length of the word α.
5. ϵ: empty word (the only word of length 0)
6. $\alpha\beta$: concatenation of α and β
7. $\alpha[i]$: i-th character of α (starting from 0)
8. $\alpha[i : j]$: subword $\alpha[i]\alpha[i + 1] \cdots \alpha[j - 1]$
9. $\alpha[: j]$: prefix of α of length j
10. $\alpha[i :]$: a suffix of α
11. $\alpha[:]$: whole word α

Occurrence of ν in η is any index i such that $\eta[i : i + |\nu|] = \nu$

Naive approach

string-searching

Given a string ν (“a needle”) and η (“a haystack”) find all occurrences of ν in η.

Search (ν, η):

1. For $i = 0, \ldots, |\eta| - |\nu| - 1$:
2. If $\eta[i : i + |\nu|] = \nu$: output i

$\nu = \textit{coconut}$

η is some very long text about coconuts

Naive approach

string-searching

Given a string ν (“a needle”) and η (“a haystack”) find all occurrences of ν in η.

Search (ν, η):

1. For $i = 0, \ldots, |\eta| - |\nu| - 1$:
2. If $\eta[i : i + |\nu|] = \nu$: output i

$\nu = \textit{coconut}$
η is some very long text about coconuts

Time complexity: $\Theta(|\nu| \cdot |\eta|)$.

Incremental algorithm

An incremental algorithm receives characters of η one by one and after receiving a new character it immediately outputs possible new occurrences of ν.

Incremental algorithm

An incremental algorithm receives characters of η one by one and after receiving a new character it immediately outputs possible new occurrences of ν.

Basic idea

We would like to remember an state. This is longest prefix of ν which is a suffix of η.

Observation: Whenever algorithm enters state ν it finds a new occurrence of ν in the input.

Assume that algorithm seen string η, is in state α and receives a new character c. How to update the state?

Incremental algorithm

An incremental algorithm receives characters of η one by one and after receiving a new character it immediately outputs possible new occurrences of ν.

Basic idea

We would like to remember an state. This is longest prefix of ν which is a suffix of η.

Observation: Whenever algorithm enters state ν it finds a new occurrence of ν in the input.

Assume that algorithm seen string η, is in state α and receives a new character c. How to update the state?

$$\text{State}(\alpha, c) = \begin{cases} \alpha c & \text{If } \alpha c \text{ is a prefix of } \nu \\ & \end{cases}$$

Incremental algorithm

An incremental algorithm receives characters of η one by one and after receiving a new character it immediately outputs possible new occurrences of ν.

Basic idea

We would like to remember an state. This is longest prefix of ν which is a suffix of η.

Observation: Whenever algorithm enters state ν it finds a new occurrence of ν in the input.

Assume that algorithm seen string η, is in state α and receives a new character c. How to update the state?

$$\text{State}(\alpha, c) = \begin{cases} \alpha c & \text{If } \alpha c \text{ is a prefix of } \nu \\ \epsilon & \\ & \end{cases}$$

Incremental algorithm

An incremental algorithm receives characters of η one by one and after receiving a new character it immediately outputs possible new occurrences of ν.

Basic idea

We would like to remember an state. This is longest prefix of ν which is a suffix of η.

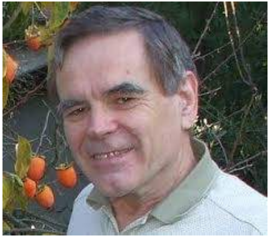
Observation: Whenever algorithm enters state ν it finds a new occurrence of ν in the input.

Assume that algorithm seen string η, is in state α and receives a new character c. How to update the state?

$$\text{State}(\alpha, c) = \begin{cases} \alpha c & \text{If } \alpha c \text{ is a prefix of } \nu \\ \epsilon & \\ \alpha' c & \alpha' c \text{ is a prefix of } \nu \text{ and } \alpha' \text{ is a suffix of } \alpha \end{cases}$$

We want to compute backward function b which tells for every prefix α of ν the longest proper suffix α' (of α) that is also a prefix of ν.

Knuth–Morris–Pratt (KMP) algorithm (1974)



Knuth–Morris–Pratt (KMP) algorithm (1974)

We want to compute backward function b which tells for every prefix α of ν the longest property suffix α' (of α) that is also a prefix of ν.

Knuth–Morris–Pratt (KMP) algorithm (1974)

Searching automaton

1. State $0, \ldots, |\nu|$
   (state s corresponds to prefix $\nu[: s]$)
2. Forward edges: $s \to s + 1$
3. Backward edges: pointing from $s > 0$ to j such that $\nu[: j]$ is a proper suffix of $\nu[: s]$

Knuth–Morris–Pratt (KMP) algorithm (1974)

Searching automaton

1. State $0, \ldots, |\nu|$
(state s corresponds to prefix $\nu[:s]$)
2. Forward edges: $s \to s+1$
3. Backward edges: pointing from $s > 0$ to j such that $\nu[:j]$ is a proper suffix of $\nu[:s]$

Step (s, c):

1. While $s \neq 0$ and $\nu[s] \neq c$:
2. $\quad s \leftarrow b[s]$.
3. If $\nu[s] = c$: $s \leftarrow s+1$.
4. Return s

Search (η, automaton for ν):

1. $s \leftarrow 0$.
2. For $i = 0, \ldots, |\eta| - 1$:
3. $\quad s \leftarrow$ Step $(s, \eta[i])$.
4. $\quad$ If $s = |\nu|$: report $i - |\nu| + 1$.

Invariant: The state s corresponds to the longest suffix of $\eta[:i]$ that is a prefix of ν.

Knuth–Morris–Pratt (KMP) algorithm (1974)

Lemma

Search will run in time $O(|\eta|)$

Proof.

1. Search itself is clearly $O(|\eta|)$

□

Step (s, c):

1. While $s \neq 0$ and $\nu[s] \neq c$:
2. $s \leftarrow b[s]$.
3. If $\nu[s] = c$: $s \leftarrow s + 1$.
4. Return s

Search (η, automaton for ν)

1. $s \leftarrow 0$.
2. For $i = 0, \ldots, |\eta| - 1$:
3. $s \leftarrow$ Step $(s, \eta[i])$.
4. If $s = |\nu|$: report $i - |\nu| + 1$.

Invariant: The state s corresponds to the longest suffix of $\eta[:i]$ that is a prefix of ν.

Knuth–Morris–Pratt (KMP) algorithm (1974)

Lemma

Search will run in time $O(|\eta|)$

Proof.

1. Search itself is clearly $O(|\eta|)$
2. Number of forward transitions in Step is $O(|\eta|)$

□

Step (s, c):

1. While $s \neq 0$ and $\nu[s] \neq c$:
2. $\quad s \leftarrow b[s]$.
3. If $\nu[s] = c$: $s \leftarrow s + 1$.
4. Return s

Search (η, automaton for ν)

1. $s \leftarrow 0$.
2. For $i = 0, \ldots, |\eta| - 1$:
3. $\quad s \leftarrow$ Step $(s, \eta[i])$.
4. $\quad$ If $s = |\nu|$: report $i - |\nu| + 1$.

Invariant: The state s corresponds to the longest suffix of $\eta[:i]$ that is a prefix of ν.

Knuth–Morris–Pratt (KMP) algorithm (1974)

Lemma

Search will run in time $O(|\eta|)$

Proof.

1. Search itself is clearly $O(|\eta|)$
2. Number of forward transitions in Step is $O(|\eta|)$
3. Number of backward transitions in Step is $O(|\eta|)$

□

Step (s, c):

1. While $s \neq 0$ and $\nu[s] \neq c$:
2. $\quad s \leftarrow b[s]$.
3. If $\nu[s] = c$: $s \leftarrow s + 1$.
4. Return s

Search (η, automaton for ν)

1. $s \leftarrow 0$.
2. For $i = 0, \dots, |\eta| - 1$:
3. $\quad s \leftarrow$ Step (s, $\eta[i]$).
4. $\quad$ If $s = |\nu|$: report $i - |\nu| + 1$.

Invariant: The state s corresponds to the longest suffix of $\eta[:i]$ that is a prefix of ν.

Knuth–Morris–Pratt (KMP) algorithm (1974)

Lemma

Search will run in time $O(|\eta|)$

Proof.

1. Search itself is clearly $O(|\eta|)$
2. Number of forward transitions in Step is $O(|\eta|)$
3. Number of backward transitions in Step is $O(|\eta|)$

□

Overall runtime $\Theta(|\eta|)$.

Step (s, c):

1. While $s \neq 0$ and $\nu[s] \neq c$:
2. $\quad s \leftarrow b[s]$.
3. If $\nu[s] = c$: $s \leftarrow s + 1$.
4. Return s

Search (η, automaton for ν)

1. $s \leftarrow 0$.
2. For $i = 0, \ldots, |\eta| - 1$:
3. $\quad s \leftarrow$ Step $(s, \eta[i])$.
4. $\quad$ If $s = |\nu|$: report $i - |\nu| + 1$.

Invariant: The state s corresponds to the longest suffix of $\eta[: i]$ that is a prefix of ν.

Knuth–Morris–Pratt (KMP) algorithm (1974)

How to obtain the automaton?

Step (s, c):

1. While $s \neq 0$ and $\nu[s] \neq c$:
2. $\quad s \leftarrow b[s]$.
3. If $\nu[s] = c$: $s \leftarrow s + 1$.
4. Return s

Search (η, automaton for ν):

1. $s \leftarrow 0$.
2. For $i = 0, \ldots, |\eta| - 1$:
3. $\quad s \leftarrow$ Step (s, $\eta[i]$).
4. $\quad$ If $s = |\nu|$: report $i - |\nu| + 1$.

Invariant: The state s corresponds to the longest suffix of $\eta[:i]$ that is a prefix of ν.

Knuth–Morris–Pratt (KMP) algorithm (1974)

How to obtain the automaton? We will steal it!

Imagine that someone has the automaton and we want to figure all backward edges

Step (s, c):

1. While $s \neq 0$ and $\nu[s] \neq c$:
2. $s \leftarrow b[s]$.
3. If $\nu[s] = c$: $s \leftarrow s + 1$.
4. Return s

Search (η, automaton for ν):

1. $s \leftarrow 0$.
2. For $i = 0, \ldots, |\eta| - 1$:
3. $s \leftarrow$ Step (s, $\eta[i]$).
4. If $s = |\nu|$: report $i - |\nu| + 1$.

Invariant: The state s corresponds to the longest suffix of $\eta[: i]$ that is a prefix of ν.

Knuth–Morris–Pratt (KMP) algorithm (1974)

How to obtain the automaton? We will steal it!

Imagine that someone has the automaton and we want to figure all backward edges

To determine $b(s)$ we need to search $\nu[1:s]$
Recall: $b(s)$ is j such that $\nu[:j]$ is the longest proper suffix of $\nu[:s]$

Step (s, c):

1. While $s \neq 0$ and $\nu[s] \neq c$:
2. $\quad s \leftarrow b[s]$.
3. If $\nu[s] = c$: $s \leftarrow s + 1$.
4. Return s

Search $(\eta$, automaton for $\nu)$:

1. $s \leftarrow 0$.
2. For $i = 0, \ldots, |\eta| - 1$:
3. $\quad s \leftarrow$ Step $(s, \eta[i])$.
4. $\quad$ If $s = |\nu|$: report $i - |\nu| + 1$.

Invariant: The state s corresponds to the longest suffix of $\eta[:i]$ that is a prefix of ν.

Knuth–Morris–Pratt (KMP) algorithm (1974)

How to obtain the automaton? We will steal it!

Imagine that someone has the automaton and we want to figure all backward edges

To determine $b(s)$ we need to search $\nu[1 : s]$
Recall: $b(s)$ is j such that $\nu[: j]$ is the longest proper suffix of $\nu[: s]$

KMPConstruction (ν):

1. $b[0] \leftarrow$undefined, $b[1] \leftarrow 0$, $s \leftarrow 0$.

Step (s, c):

1. While $s \neq 0$ and $\nu[s] \neq c$:
2. $s \leftarrow b[s]$.
3. If $\nu[s] = c$: $s \leftarrow s + 1$.
4. Return s

Search (η, automaton for ν):

1. $s \leftarrow 0$.
2. For $i = 0, \ldots, |\eta| - 1$:
3. $s \leftarrow$ Step (s, $\eta[i]$).
4.   If $s = |\nu|$: report $i - |\nu| + 1$.

Invariant: The state s corresponds to the longest suffix of $\eta[: i]$ that is a prefix of ν.

Knuth–Morris–Pratt (KMP) algorithm (1974)

How to obtain the automaton? We will steal it!

Imagine that someone has the automaton and we want to figure all backward edges

To determine $b(s)$ we need to search $\nu[1 : s]$
Recall: $b(s)$ is j such that $\nu[: j]$ is the longest proper suffix of $\nu[: s]$

KMPConstruction (ν):

1. $b[0] \leftarrow$undefined, $b[1] \leftarrow 0$, $s \leftarrow 0$.
2. For $i = 2, \ldots, |\nu|$:
3. $\quad s \leftarrow$ Step $(s, \nu[i-1])$.
4. $\quad b[i] \leftarrow s$.

Step (s, c):

1. While $s \neq 0$ and $\nu[s] \neq c$:
2. $\quad s \leftarrow b[s]$.
3. If $\nu[s] = c$: $s \leftarrow s + 1$.
4. Return s

Search (η, automaton for ν):

1. $s \leftarrow 0$.
2. For $i = 0, \ldots, |\eta| - 1$:
3. $\quad s \leftarrow$ Step $(s, \eta[i])$.
4. $\quad$If $s = |\nu|$: report $i - |\nu| + 1$.

Invariant: The state s corresponds to the longest suffix of $\eta[: i]$ that is a prefix of ν.

Knuth–Morris–Pratt (KMP) algorithm (1974)

How to obtain the automaton? We will steal it!

Imagine that someone has the automaton and we want to figure all backward edges

To determine $b(s)$ we need to search $\nu[1 : s]$
Recall: $b(s)$ is j such that $\nu[: j]$ is the longest proper suffix of $\nu[: s]$

KMPConstruction (ν):

1. $b[0] \leftarrow$undefined, $b[1] \leftarrow 0$, $s \leftarrow 0$.
2. For $i = 2, \ldots, |\nu|$:
3. $\quad s \leftarrow$ Step $(s, \nu[i-1])$.
4. $\quad b[i] \leftarrow s$.

Step (s, c):

1. While $s \neq 0$ and $\nu[s] \neq c$:
2. $\quad s \leftarrow b[s]$.
3. If $\nu[s] = c$: $s \leftarrow s + 1$.
4. Return s

Search (η, automaton for ν):

1. $s \leftarrow 0$.
2. For $i = 0, \ldots, |\eta| - 1$:
3. $\quad s \leftarrow$ Step $(s, \eta[i])$.
4. $\quad$If $s = |\nu|$: report $i - |\nu| + 1$.

Theorem

Algorithm KMP will finish in time $\Theta(|\eta| + |\nu|)$.

Invariant: The state s corresponds to the longest suffix of $\eta[: i]$ that is a prefix of ν.

Rabin–Karp algorithm (1987)

Recall the rotating/sliding hash function

$$H(x_1, x_2, \ldots, x_K) = (x_1 P^{K-1} + x_2 P^{K-2} + \cdots + x_{K-1} P^1 + x_K P^0) \mod N$$

For $K = |\nu|$, prime number P and $N > 0$.

Rabin–Karp algorithm (1987)

Recall the rotating/sliding hash function

$$H(x_1, x_2, \ldots, x_K) = (x_1 P^{K-1} + x_2 P^{K-2} + \cdots + x_{K-1} P^1 + x_K P^0) \mod N$$

For $K = |\nu|$, prime number P and $N > 0$. Observe that:

$$\begin{aligned} H(x_2, x_3, \ldots, x_{K+1}) &= (x_2 P^{K-1} + x_3 P^{K-2} + \cdots + x_K P^1 + x_{K+1} P^0) \mod N \\ &= (P \cdot H(x_1, x_2, \ldots, x_K) - x_1 P^K + x_{K+1}) \mod N \end{aligned}$$

Rabin–Karp algorithm (1987)

Recall the rotating/sliding hash function

$$H(x_1, x_2, \ldots, x_K) = (x_1 P^{K-1} + x_2 P^{K-2} + \cdots + x_{K-1} P^1 + x_K P^0) \mod N$$

For $K = |\nu|$, prime number P and $N > 0$. Observe that:

$$\begin{aligned} H(x_2, x_3, \ldots, x_{K+1}) &= (x_2 P^{K-1} + x_3 P^{K-2} + \cdots + x_K P^1 + x_{K+1} P^0) \mod N \\ &= (P \cdot H(x_1, x_2, \ldots, x_K) - x_1 P^K + x_{K+1}) \mod N \end{aligned}$$

RabinKarpSearch (ν, η):

1. Choose prime P and $N > 1$.
2. Precompute $P^{|\nu|} \mod N$.
3. $n \leftarrow H(\nu)$.
4. $h \leftarrow H(\eta[: |\nu|])$.
5. For $i = 0, \ldots |\eta| - |\nu| - 1$:
6.   if $h = n$ and $\eta[i : i + |\nu|] = \nu$: report i.
7. $h \leftarrow (P \cdot h - \eta[i] \cdot P^{|\nu|} + \eta[i + K]) \mod N$.

Rabin–Karp algorithm (1987)

Recall the rotating/sliding hash function

$$H(x_1, x_2, \ldots, x_K) = (x_1 P^{K-1} + x_2 P^{K-2} + \cdots + x_{K-1} P^1 + x_K P^0) \mod N$$

For $K = |\nu|$, prime number P and $N > 0$. Observe that:

$$\begin{aligned} H(x_2, x_3, \ldots, x_{K+1}) &= (x_2 P^{K-1} + x_3 P^{K-2} + \cdots + x_K P^1 + x_{K+1} P^0) \mod N \\ &= (P \cdot H(x_1, x_2, \ldots, x_K) - x_1 P^K + x_{K+1}) \mod N \end{aligned}$$

RabinKarpSearch (ν, η):

1. Choose prime P and $N > 1$.
2. Precompute $P^{|\nu|} \mod N$.
3. $n \leftarrow H(\nu)$.
4. $h \leftarrow H(\eta[: |\nu|])$.
5. For $i = 0, \ldots |\eta| - |\nu| - 1$:
6. if $h = n$ and $\eta[i : i + |\nu|] = \nu$: report i.
7. $h \leftarrow (P \cdot h - \eta[i] \cdot P^{|\nu|} + \eta[i + K]) \mod N$.

Expected time complexity: $O(|\nu| + |\eta| + C|\nu| + |\eta|/N \cdot |\nu|)$ where C is number of occurences reported.