String Matching

Dr Timothy Kimber

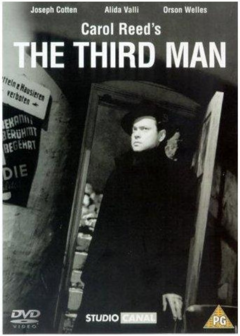
February 2015

String Matching

Given the text

Like the fella says, in Italy for thirty years under the Borgias they had warfare, terror, murder, and bloodshed, but they produced Michelangelo, Leonardo da Vinci, and the Renaissance. In Switzerland they had brotherly love - they had five hundred years of democracy and peace, and what did that produce? The cuckoo clock.

– Harry Lime (The Third Man)



Where does the pattern "they" occur?

String Matching

- The pattern and the text are both strings
- A string is any sequence of characters from some alphabet $\mathcal{A}$
- Used in document search, virus detection, gene sequencing etc.

Definition (Shift)

Given two sequences $P = \langle p_1, \ldots, p_M \rangle$ and $T = \langle t_1, \ldots, t_N \rangle$,
P occurs with shift S in T iff $t_{i+S} = p_i$ for all $1 \leq i \leq M$.

Problem *(String Match)*

Input: a sequence P of characters $\langle p_1, \ldots, p_M \rangle$

Input: a sequence T of characters $\langle t_1, \ldots, t_N \rangle$

Output: all shifts with which P occurs in T

Example

Using the alphabet $\{a, b, c\}$

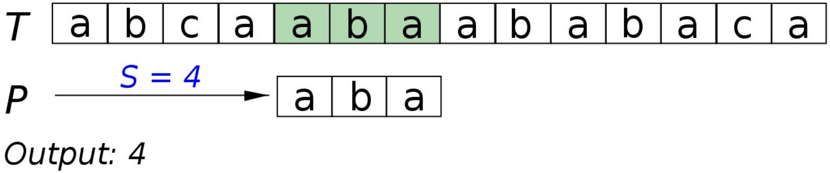
T	a	b	c	a	a	b	a	a	b	a	b	a	c	a

P	a	b	a

- $M = 3$, $N = 14$
- The minimum shift is 0
- The maximum shift is $N - M$
- Matches for this example at $S = 4, 7, 9$

Example

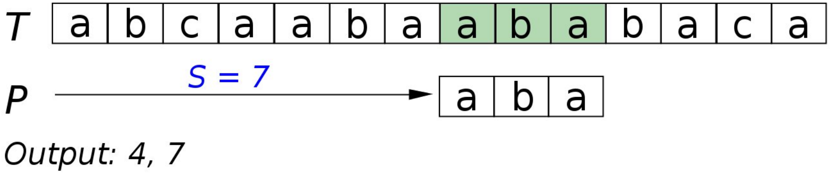
Using the alphabet $\{a, b, c\}$



- $M = 3,\ N = 14$
- The minimum shift is 0
- The maximum shift is $N - M$
- Matches for this example at $S = 4, 7, 9$

Example

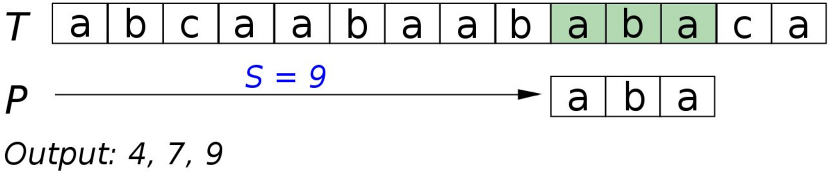
Using the alphabet $\{a, b, c\}$



- $M = 3$, $N = 14$
- The minimum shift is 0
- The maximum shift is $N - M$
- Matches for this example at $S = 4, 7, 9$

Example

Using the alphabet $\{a, b, c\}$



- $M = 3$, $N = 14$
- The minimum shift is 0
- The maximum shift is $N - M$
- Matches for this example at $S = 4, 7, 9$

Naive Algorithm

As a starting point we consider a naive approach

Naive Match (Input: $P = \langle p_1, \ldots, p_M \rangle$, $T = \langle t_1, \ldots, t_N \rangle$)

- For $S = 0$ to $N - M$
  - If $\langle t_{1+S}, \ldots, t_{M+S} \rangle = \langle p_1, \ldots, p_S \rangle$
    - Output S
- HALT

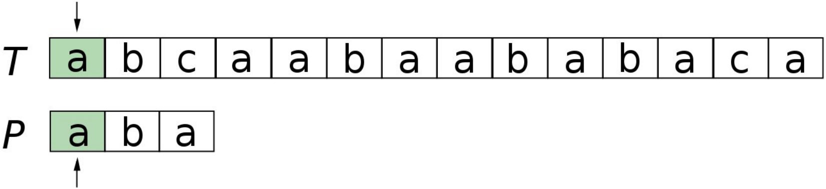
- P is compared with $\langle t_{1+S}, \ldots, t_{M+S} \rangle$ for each possible shift

Questions

- How should the string equality check be implemented?
- What is the time complexity?
- What are the best and worst cases, and their complexity?

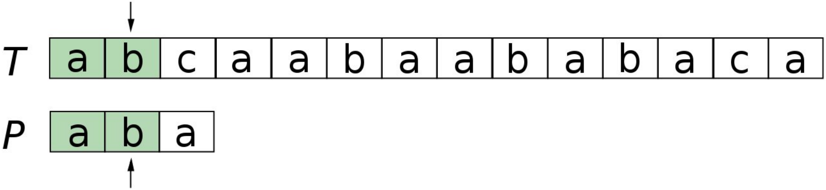
Example

Look at the string matching in detail



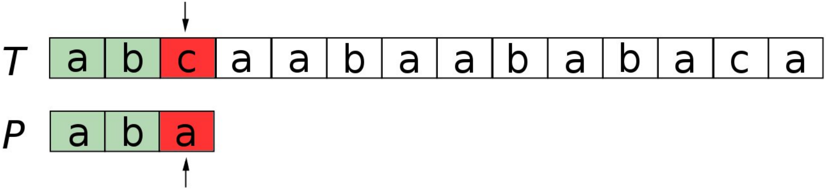
Example

Look at the string matching in detail



Example

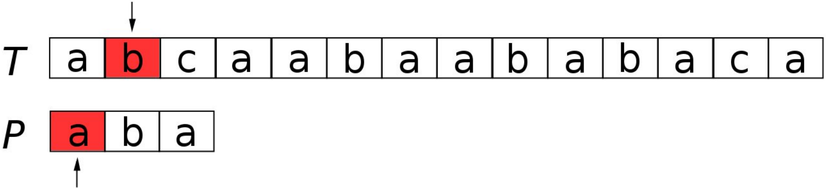
Look at the string matching in detail



- What happens when the match fails?
- The text pointer returns to $S + 1$
- This character was already looked at

Example

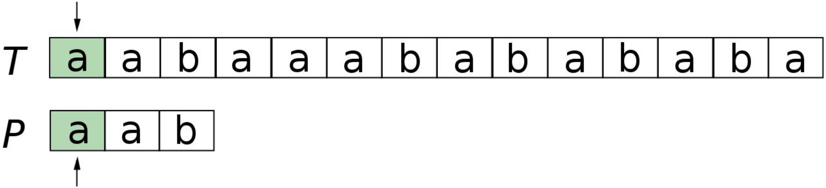
Look at the string matching in detail



- What happens when the match fails?
- The text pointer returns to $S + 1$
- This character was already looked at

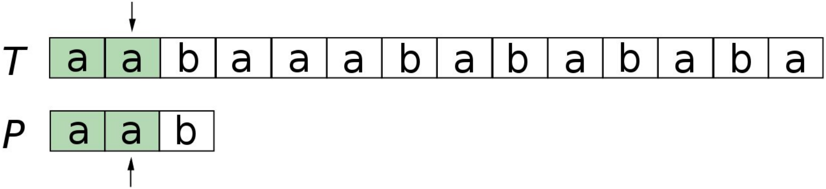
Example

Can we design a linear algorithm in which we look at each text char once?



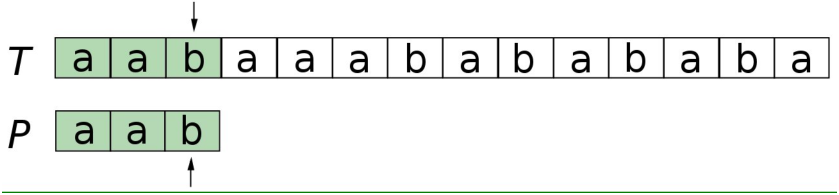
Example

Can we design a linear algorithm in which we look at each text char once?



Example

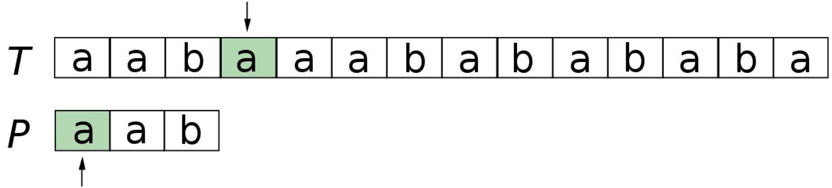
Can we design a linear algorithm in which we look at each text char once?



Output: 0

Example

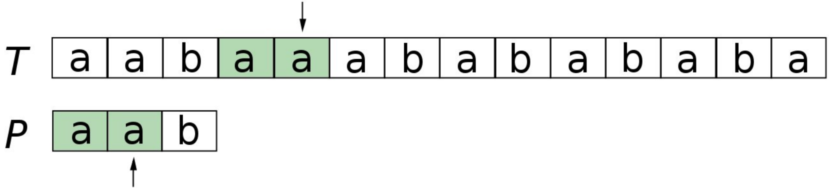
Can we design a linear algorithm in which we look at each text char once?



Output: 0

Example

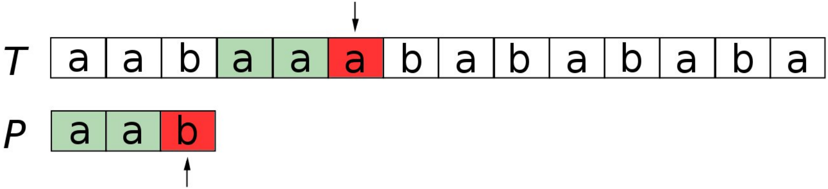
Can we design a linear algorithm in which we look at each text char once?



Output: 0

Example

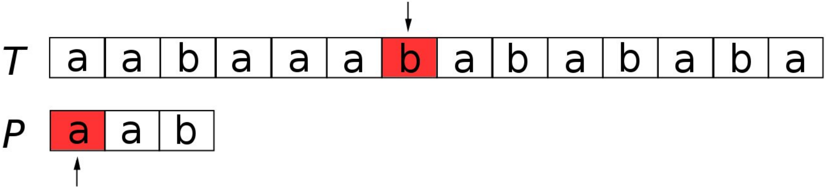
Can we design a linear algorithm in which we look at each text char once?



Output: 0

Example

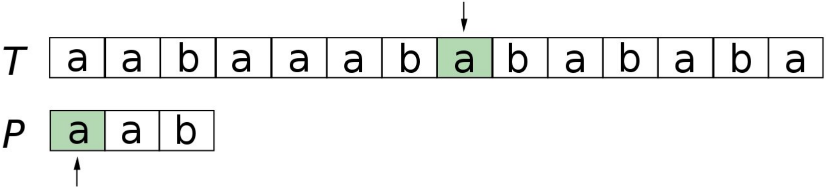
Can we design a linear algorithm in which we look at each text char once?



Output: 0

Example

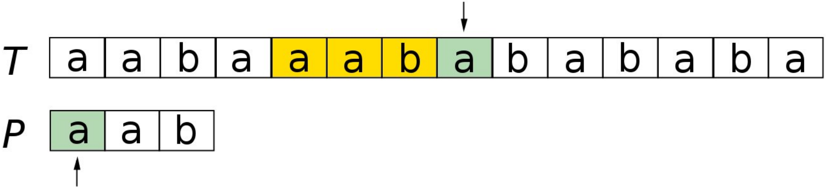
Can we design a linear algorithm in which we look at each text char once?



Output: 0

Example

Can we design a linear algorithm in which we look at each text char once?

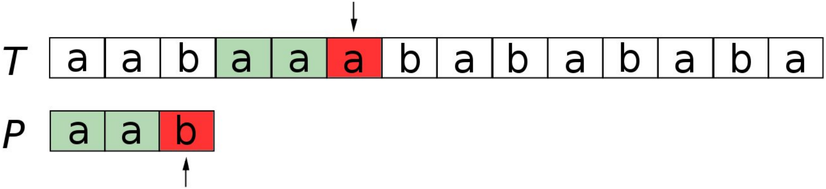


Output: 0

- The match at $S = 4$ was missed
- What happened?

Example

Can we design a linear algorithm in which we look at each text char once?

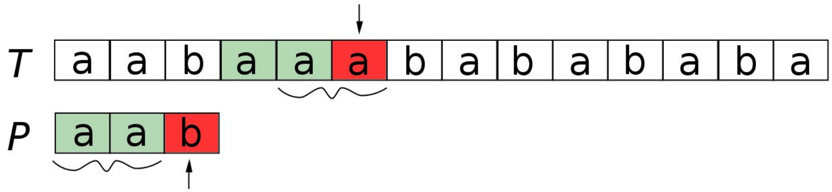


Output: 0

- There is no match at $S = 3$
- However, a prefix of P has been matched
- There might be a match at $S = 4$
- Going back to the beginning of P was wrong

Example

Can we design a linear algorithm in which we look at each text char once?



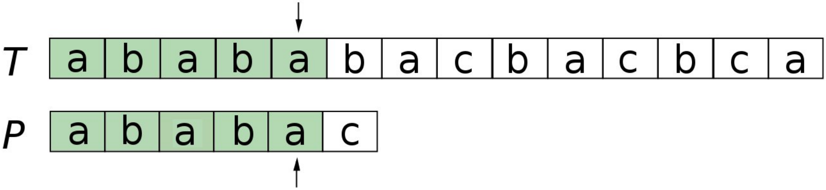
Output: 0

- There is no match at $S = 3$
- However, a prefix of P has been matched
- There might be a match at $S = 4$
- Going back to the beginning of P was wrong

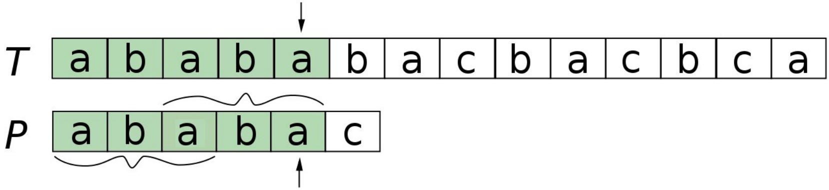
Another Example

T	a	b	a	b	a	b	a	c	b	a	c	b	c	a
P	a	b	a	b	a	c								

Another Example

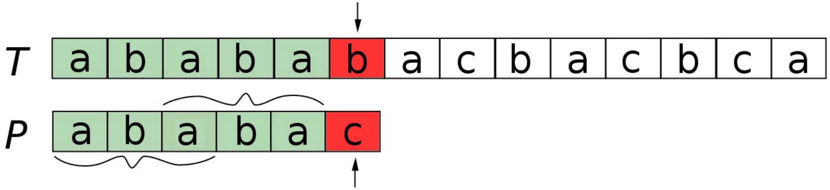


Another Example



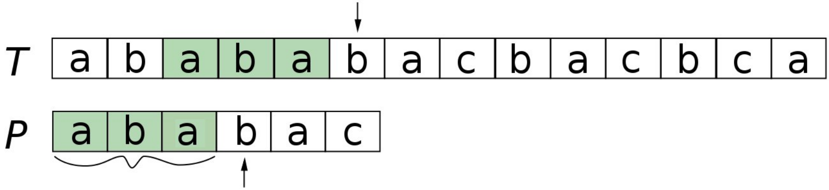
- There is a suffix of the matched text that is a prefix of P

Another Example



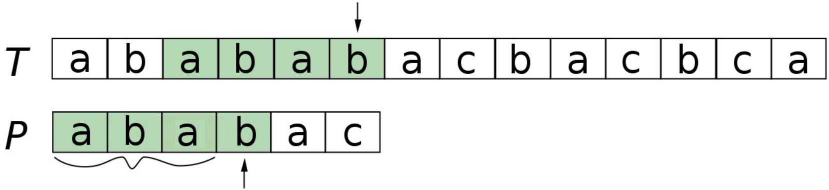
- There is a suffix of the matched text that is a prefix of P
- Proceed starting from the matched prefix
- Need to identify such subpatterns in P

Another Example



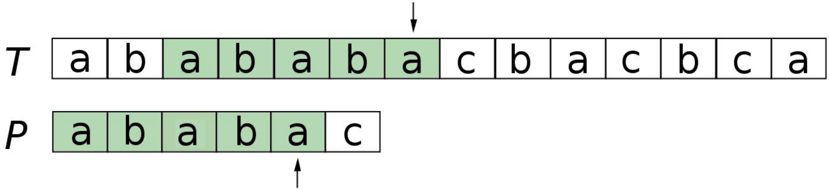
- There is a suffix of the matched text that is a prefix of P
- Proceed starting from the matched prefix
- Need to identify such subpatterns in P

Another Example



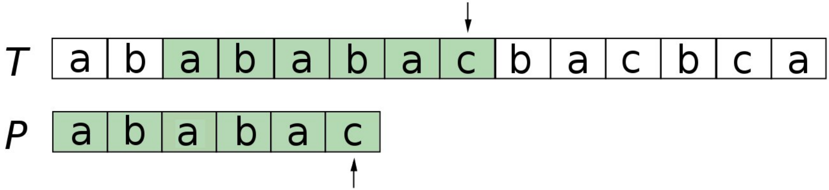
- There is a suffix of the matched text that is a prefix of P
- Proceed starting from the matched prefix
- Need to identify such subpatterns in P

Another Example



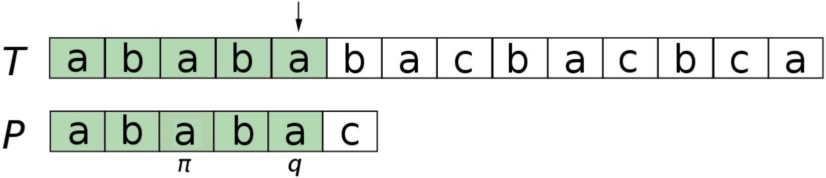
- There is a suffix of the matched text that is a prefix of P
- Proceed starting from the matched prefix
- Need to identify such subpatterns in P

Another Example



- There is a suffix of the matched text that is a prefix of P
- Proceed starting from the matched prefix
- Need to identify such subpatterns in P

Knuth–Morris–Pratt



- $\langle p_1, \ldots, p_q \rangle$ is the prefix of P matched so far
- Find the longest prefix of P that is a suffix of $\langle p_2, \ldots, p_q \rangle$
- So, find $0 \leq \pi < q$ such that $\langle p_1, \ldots, p_\pi \rangle = \langle p_{q-\pi+1}, \ldots, p_q \rangle$
- And there is no $\pi' > \pi$
- After non-match restart from $q = \pi$

Computing the Prefix

The prefixes are a property of the pattern P

- Each q will have a different π
- This prefix function $\pi(q)$ can be precomputed without referring to T
- We can store π in a sequence of length M

P	a	b	a	b	a	c
π	0	0	1	2	3	0

Knuth–Morris–Pratt

KMP (Input: $P = \langle p_1, \ldots, p_M \rangle$, $T = \langle t_1, \ldots, t_N \rangle$)

- π = Compute-Prefixes P
- $q = 0$ <-- characters matched so far
- For $i = 1$ to N
  - While $q > 0$ and $t_i \neq p_{q+1}$ <-- no match, reset using π
    - $q = \pi_q$
  - If $t_i = p_{q+1}$
    - $q = q + 1$
  - If $q = M$
    - Output $i - M$
    - $q = \pi_q$ <-- full match, reset using π
- HALT

Knuth–Morris–Pratt

Compute-Prefixes (Input: $P = \langle p_1, \ldots, p_M \rangle$)

- $\pi_1 = 0$
- $k = 0$ <-- largest prefix found
- For $q = 2$ to M
  - While $k > 0$ and $p_q \neq p_{k+1}$ <-- no match, reset using π
    - $k = \pi_k$
  - If $p_q = p_{k+1}$ <-- next char matches char after prefix
    - $k = k + 1$
  - $\pi_q = k$
- Return π and HALT

Computing π

P	a	b	a	b	a	c
π						

- k is the length of the current prefix
- Check if the next char extends the prefix or not

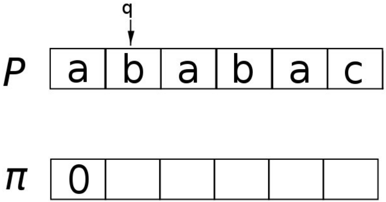
Computing π

P	a	b	a	b	a	c

π	0					

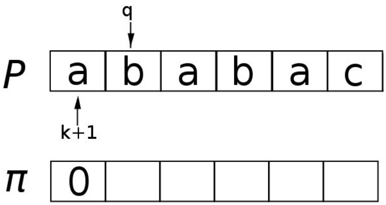
- k is the length of the current prefix
- Check if the next char extends the prefix or not

Computing π



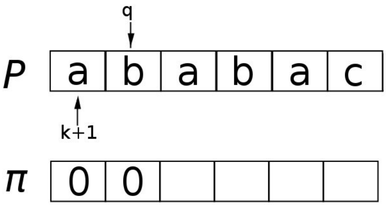
- k is the length of the current prefix
- Check if the next char extends the prefix or not

Computing π



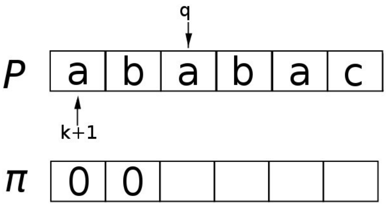
- k is the length of the current prefix
- Check if the next char extends the prefix or not

Computing π



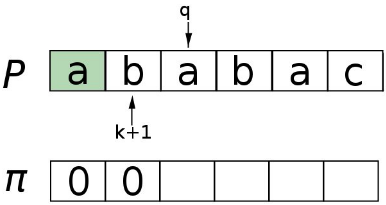
- k is the length of the current prefix
- Check if the next char extends the prefix or not

Computing π



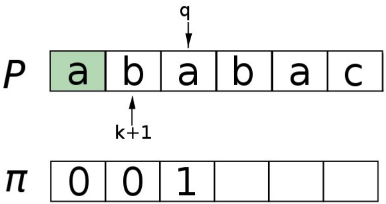
- k is the length of the current prefix
- Check if the next char extends the prefix or not

Computing π



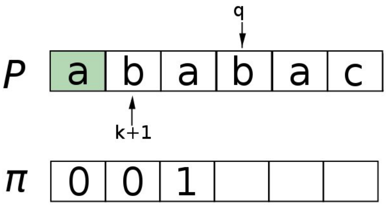
- k is the length of the current prefix
- Check if the next char extends the prefix or not

Computing π



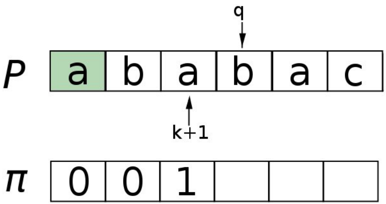
- k is the length of the current prefix
- Check if the next char extends the prefix or not

Computing π



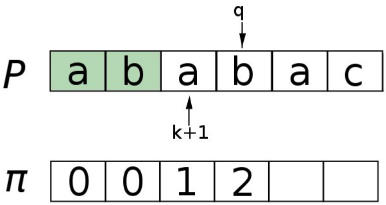
- k is the length of the current prefix
- Check if the next char extends the prefix or not

Computing π



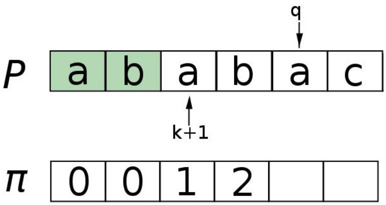
- k is the length of the current prefix
- Check if the next char extends the prefix or not

Computing π



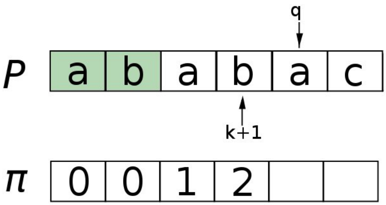
- k is the length of the current prefix
- Check if the next char extends the prefix or not

Computing π



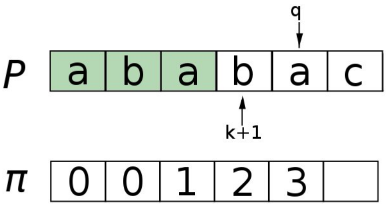
- k is the length of the current prefix
- Check if the next char extends the prefix or not

Computing π



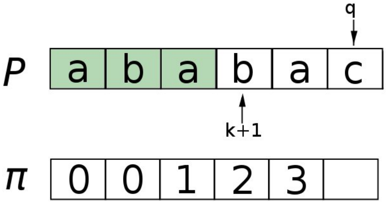
- k is the length of the current prefix
- Check if the next char extends the prefix or not

Computing π



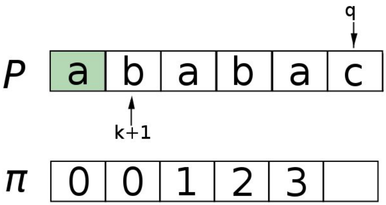
- k is the length of the current prefix
- Check if the next char extends the prefix or not

Computing π



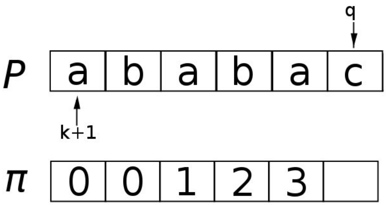
- k is the length of the current prefix
- Check if the next char extends the prefix or not

Computing π



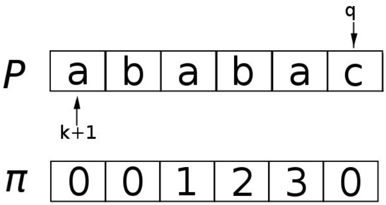
- k is the length of the current prefix
- Check if the next char extends the prefix or not

Computing π



- k is the length of the current prefix
- Check if the next char extends the prefix or not

Computing π



- k is the length of the current prefix
- Check if the next char extends the prefix or not

Performance of Compute-Prefixes

Compute-Prefixes (Input: $P = \langle p_1, \ldots, p_M \rangle$)

- $\pi_1 = 0$
- $k = 0$ <-- largest prefix found
- For $q = 2$ to M
  - While $k > 0$ and $p_q \neq p_{k+1}$ <-- no match, reset using π
    - $k = \pi_k$
  - If $p_q = p_{k+1}$ <-- next char matches char after prefix
    - $k = k + 1$
  - $\pi_q = k$
- Return π and HALT

- The while loop is the key to analysing performance
- The total increase in k is $\leq m - 1$ (max $+1$ per iteration)
- The while loop must decrease k since $k < q$ and so $\pi_k < k$
- So, the while loop runs maximum of $m - 1$ times, total

Performance

KMP (Input: $P = \langle p_1, \ldots, p_M \rangle$, $T = \langle t_1, \ldots, t_N \rangle$)

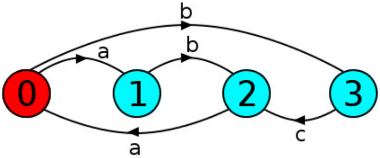
- π = Compute-Prefixes P
- $q = 0$ <-- characters matched so far
- For $i = 1$ to N
    - While $q > 0$ and $t_i \neq p_{q+1}$ <-- no match, reset using π
        - $q = \pi_q$
    - If $t_i = p_{q+1}$
        - $q = q + 1$
    - If $q = M$
        - Output $i - M$
        - $q = \pi_q$ <-- full match, reset using π
- HALT

- Similar analysis to Compute-Prefixes
- Total increase in q is $\leq N$, so $T(N) = \Theta(N)$

Finite Automata

The KMP algorithm is an optimisation of a finite automaton

- We have a set of states
- We have a set of events — occurrences of characters
- Each event changes the current state



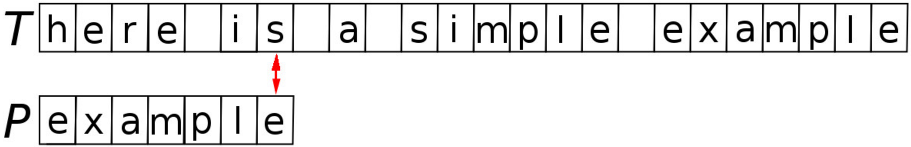
A simpler automaton would define a separate event for every possible character in every state

- This takes $O(m|\mathcal{A}|)$-time for preprocessing

Better Again?

KMP examines every character in T. Can we do even better?

- We would have to skip some of the text

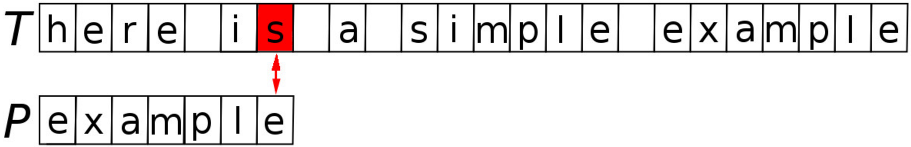


- Start matching at the right of the pattern
- If the text character not in P then shift past it

Better Again?

KMP examines every character in T. Can we do even better?

- We would have to skip some of the text

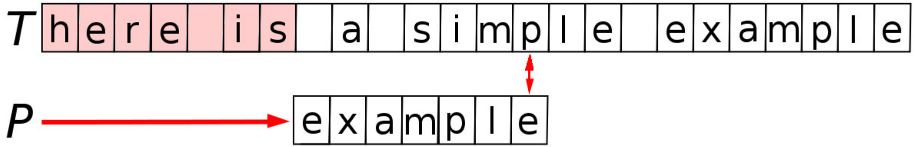


- Start matching at the right of the pattern
- If the text character not in P then shift past it

Better Again?

KMP examines every character in T. Can we do even better?

- We would have to skip some of the text

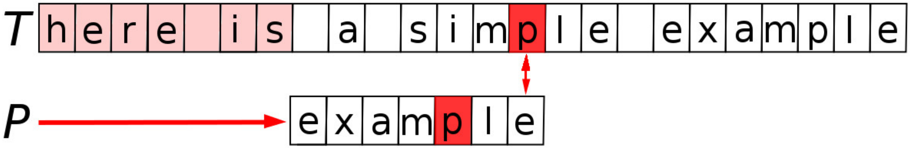


- Start matching at the right of the pattern
- If the text character not in P then shift past it

Bad Characters

If there is a bad character β in T, then shift

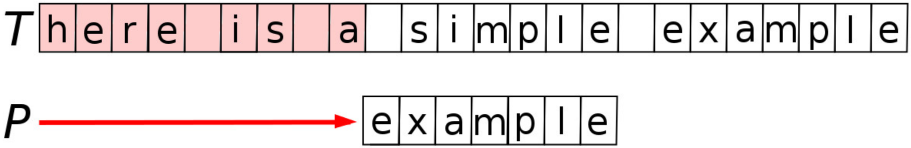
- so that P skips β, if β is not in P
- to align β with its rightmost occurrence, if β is in P
- to align a prefix of P with a suffix of the current match in T



Bad Characters

If there is a bad character β in T, then shift

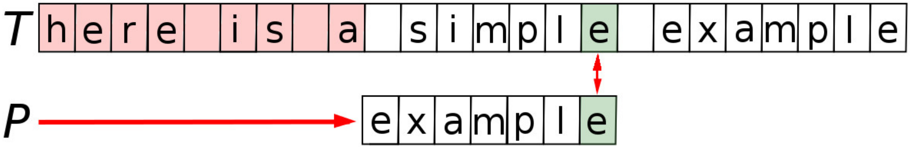
- so that P skips β, if β is not in P
- to align β with its rightmost occurrence, if β is in P
- to align a prefix of P with a suffix of the current match in T



Bad Characters

If there is a bad character β in T, then shift

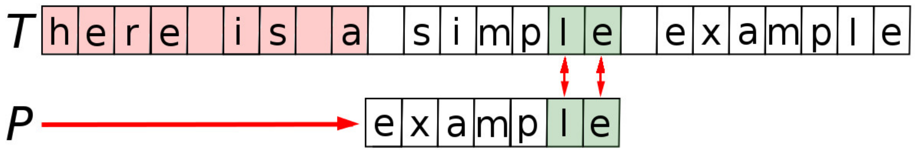
- so that P skips β, if β is not in P
- to align β with its rightmost occurrence, if β is in P
- to align a prefix of P with a suffix of the current match in T



Bad Characters

If there is a bad character β in T, then shift

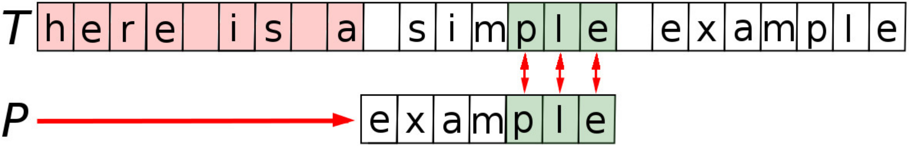
- so that P skips β, if β is not in P
- to align β with its rightmost occurrence, if β is in P
- to align a prefix of P with a suffix of the current match in T



Bad Characters

If there is a bad character β in T, then shift

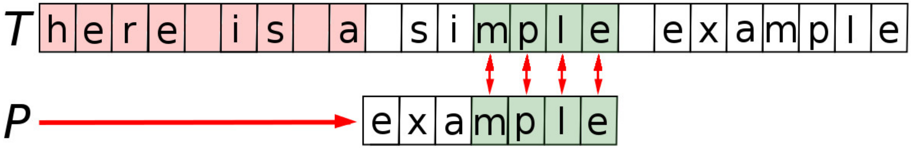
- so that P skips β, if β is not in P
- to align β with its rightmost occurrence, if β is in P
- to align a prefix of P with a suffix of the current match in T



Bad Characters

If there is a bad character β in T, then shift

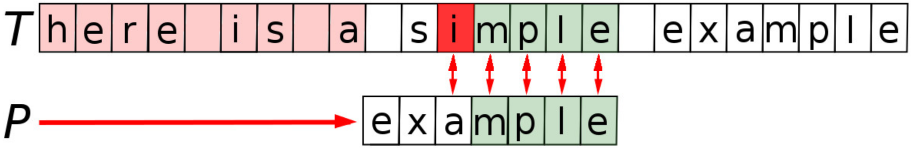
- so that P skips β, if β is not in P
- to align β with its rightmost occurrence, if β is in P
- to align a prefix of P with a suffix of the current match in T



Bad Characters

If there is a bad character β in T, then shift

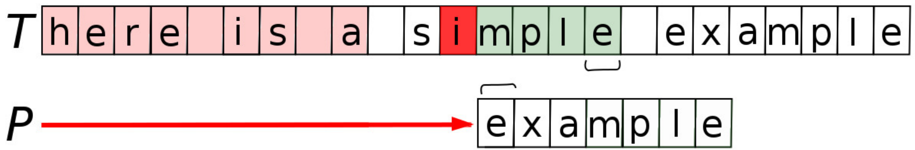
- so that P skips β, if β is not in P
- to align β with its rightmost occurrence, if β is in P
- to align a prefix of P with a suffix of the current match in T



Bad Characters

If there is a bad character β in T, then shift

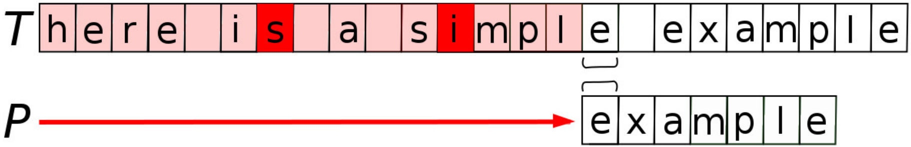
- so that P skips β, if β is not in P
- to align β with its rightmost occurrence, if β is in P
- to align a prefix of P with a suffix of the current match in T



Bad Characters

If there is a bad character β in T, then shift

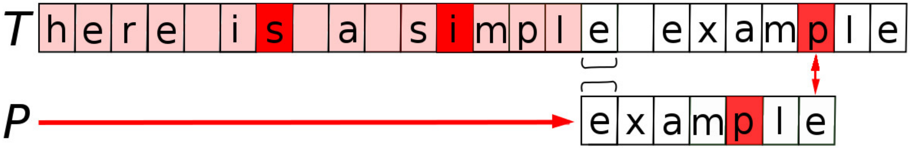
- so that P skips β, if β is not in P
- to align β with its rightmost occurrence, if β is in P
- to align a prefix of P with a suffix of the current match in T



Bad Characters

If there is a bad character β in T, then shift

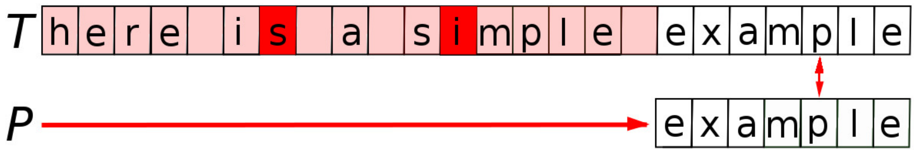
- so that P skips β, if β is not in P
- to align β with its rightmost occurrence, if β is in P
- to align a prefix of P with a suffix of the current match in T



Bad Characters

If there is a bad character β in T, then shift

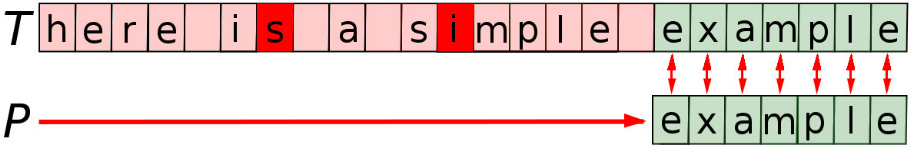
- so that P skips β, if β is not in P
- to align β with its rightmost occurrence, if β is in P
- to align a prefix of P with a suffix of the current match in T



Bad Characters

If there is a bad character β in T, then shift

- so that P skips β, if β is not in P
- to align β with its rightmost occurrence, if β is in P
- to align a prefix of P with a suffix of the current match in T



Tactics

The Boyer–Moore algorithm uses the tactics shown above

- The aim is to shift P as far right as possible
- This minimizes the number of comparisons required

The bad character rule (BCR):

- next character in T does not match next character in P

The good suffix rule (GSR):

- a suffix of P has been matched up to a bad character in T

Shift the maximum amount indicated by BCR or GSR

- shift only depends on P, so can be precomputed

Bad Character Rule

Candidate actions if the rule is satisfied

- Shift P to the right-most occurrence (RMO) of β within P, or
- Shift P past β if it is not in P

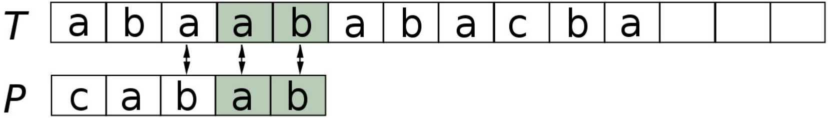
Compute RMO(Input: $P = [P_1, \ldots, P_M]$)

- For $i = 1$ to $|\mathcal{A}|$
  - $occ[i] = 0$
- For $j = 1$ to M
  - $occ[P[j]] = j$

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s
3	0	0	0	7	0	0	0	0	0	0	6	4	0	0	5	0	0	0

Bad Use of Bad Character Rule

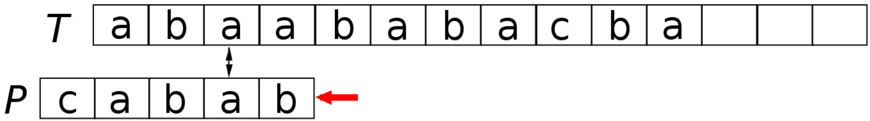
Sometimes the bad character tactic fails



- It can produce a negative shift

Bad Use of Bad Character Rule

Sometimes the bad character tactic fails

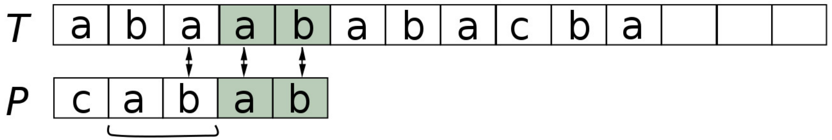


- It can produce a negative shift

Good Suffix Rule

Actions for the GSR:

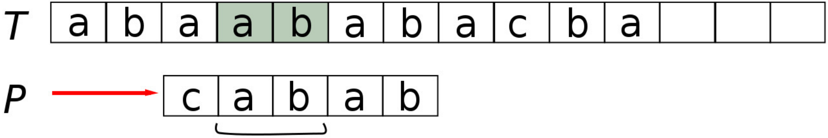
- **Case 1** If there is another occurrence of the whole suffix in P, shift P so the previous occurrence is aligned with the match in T
- **Case 2** Otherwise, if a suffix of the match is a prefix of P, shift P so the largest such prefix is aligned with its match in T
- **Case 3** Otherwise (no part of the matched suffix occurs in P), shift P past the match



Good Suffix Rule

Actions for the GSR:

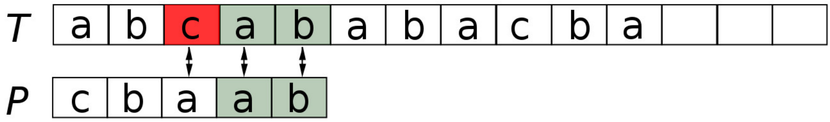
- **Case 1** If there is another occurrence of the whole suffix in P, shift P so the previous occurrence is aligned with the match in T
- **Case 2** Otherwise, if a suffix of the match is a prefix of P, shift P so the largest such prefix is aligned with its match in T
- **Case 3** Otherwise (no part of the matched suffix occurs in P), shift P past the match



Good Suffix Rule

Actions for the GSR:

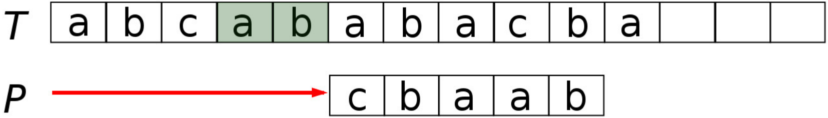
- Case 1 If there is another occurrence of the whole suffix in P, shift P so the previous occurrence is aligned with the match in T
- Case 2 Otherwise, if a suffix of the match is a prefix of P, shift P so the largest such prefix is aligned with its match in T
- Case 3 Otherwise (no part of the matched suffix occurs in P), shift P past the match



Good Suffix Rule

Actions for the GSR:

- Case 1 If there is another occurrence of the whole suffix in P, shift P so the previous occurrence is aligned with the match in T
- Case 2 Otherwise, if a suffix of the match is a prefix of P, shift P so the largest such prefix is aligned with its match in T
- Case 3 Otherwise (no part of the matched suffix occurs in P), shift P past the match



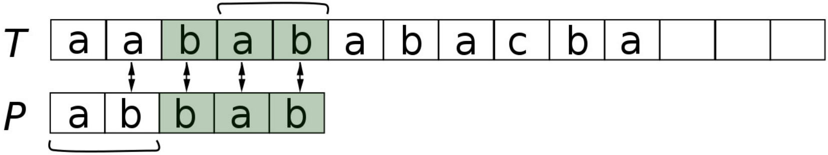
Good Suffix Rule

Actions for the GSR:

Case 1 If there is another occurrence of the whole suffix in P, shift P so the previous occurrence is aligned with the match in T

Case 2 Otherwise, if a suffix of the match is a prefix of P, shift P so the largest such prefix is aligned with its match in T

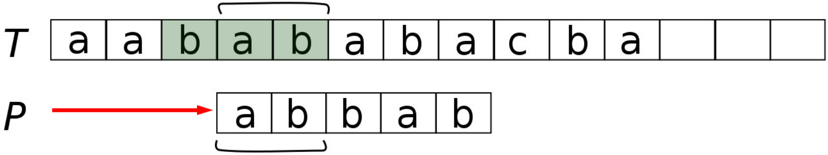
Case 3 Otherwise (no part of the matched suffix occurs in P), shift P past the match



Good Suffix Rule

Actions for the GSR:

- Case 1 If there is another occurrence of the whole suffix in P, shift P so the previous occurrence is aligned with the match in T
- Case 2 Otherwise, if a suffix of the match is a prefix of P, shift P so the largest such prefix is aligned with its match in T
- Case 3 Otherwise (no part of the matched suffix occurs in P), shift P past the match



Boyer–Moore Algorithm

- BCR results in *occ* storing the RMOs of each character in P
- GSR results in s storing the shift distance if a mismatch occurs at $P[i]$

procedure $\textsc{Boyer–Moore}(P = [p_1, \ldots, p_M],\ T = [t_1, \ldots, t_N])$
    $i = 1$
    while $i \le N - M + 1$ **do** ▷ scan T from left to right
        $j = M$ ▷ scan P from right to left
        while $j \ge 1$ and $P[j] == T[i + j - 1]$ **do**
            $j = j - 1$
        if $j < 1$ **then** ▷ full match
            Output i
            $i = i + s[1]$
        else ▷ mismatch for $T[i + j - 1]$
            $i = i + \textsc{Max}(s[j], j - occ[T[i + j - 1]])$

Performance

- Preprocessing time is $\Theta(M)$
- In general, matching time is $O(N \times M)$
- In worst case matching is $\Theta(N \times M)$
- In practice, matching in natural language texts, or when $|\mathcal{A}| >> M$ matching time is $\Omega(N/M)$
- This lower bound is possible because M characters may be skipped