Algorithms and Data Structures

Lecture 12

String Matching (Text Search)

Fabian Kuhn

Algorithms and Complexity

Text Search / String Matching

Given:

- two strings
- text T (typically long)
- pattern P (typically short)

Goal:

- Find all occurrences of P in T

Assumptions:

- Length of text T : $\boldsymbol{n}$,   Length of pattern P : $\boldsymbol{m}$ $(\boldsymbol{m} \ll \boldsymbol{n})$

Example:

- Search pattern $P =$"ABCA" in the following string

$T =$ `ABI CLABCAD LHABCABCA KAHBCA ALBCABABCABL LKAGA`

Motivation

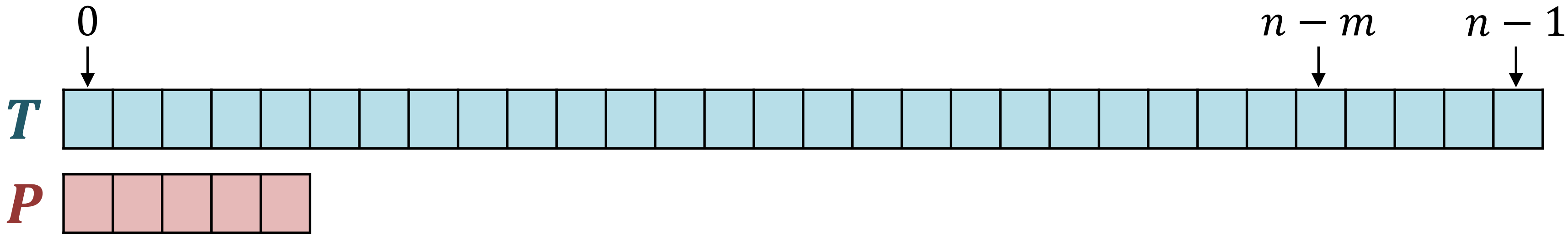
- This is obviously important...

- Required in every text editor
  - Every editor has a find function

- Supported by higher programming languages:
  - Java: String.indexOf(String pattern, int fromThisPosition)
  - C++: std::string.find(std::string str, size_t fromThisPosition)
  - Python: str.find(pattern, from), where str is a string

Naïve Algorithm

- Go through the text from left to right
- The pattern can occur at each of the positions $s = 0, \dots, n - m$



- Test at each of these positions if there is a match between the pattern an the corresponding part of the text,
  - by going trough the pattern character by character and comparing with the corresponding character in the text.

Naïve Algorithm

```
TestPosition($s$):        // tests if $T[s, \dots, s+m-1] == P$
    $t = 0$
    while $t < m$ and $T[s+t] == P[t]$ do
        $t = t + 1$
    return $(t == m)$
```

Laufzeit:

$$\#\text{Iter.} := \begin{cases} m, & \text{if } P \text{ is found} \\ 1 + \min\limits_{0<i<m} T[s+i] \neq P[i], & \text{else} \end{cases}$$

- Worst Case: $O(m)$
  - In the worst case, one has to check all m positions of P
  - This is in particular the case if P is found
- Best Case: $O(1)$
  - In the best case, we already see at the first character that there is no match (is $T[s] \neq P[0]$)

Naïve Algorithm

```
TestPosition($s$):        // tests if $T[s, \dots, s+m-1] == P$
    $t = 0$
    while $t < m$ and $T[s+t] == P[t]$ do
        $t = t + 1$
    return $(t == m)$
```

```
String-Matching:
    for $s$ from $0$ to $n - m$ do
        if TestPosition($s$) then
            report found match at position $s$
```

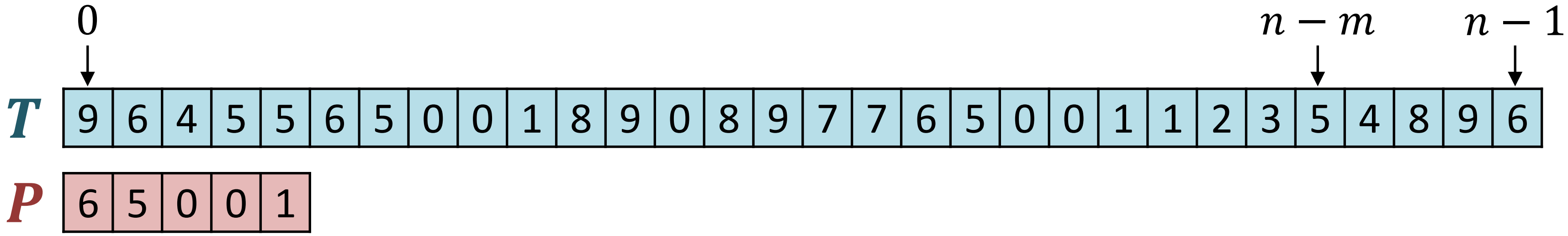
Running Time:

- Worst Case: $O(n \cdot m)$
- Best Case : $O(n)$

Rabin-Karp Algorithm

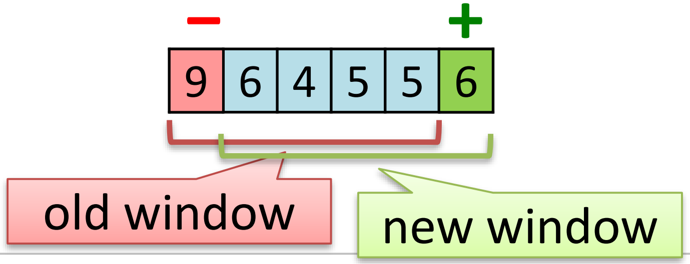
Basic Idea

- For simplicity , we assume that the text only consists of the digits $0, \dots, 9$
  - Then we can understand the pattern and the text window as numbers
- We again move a window of length m over the text and check at each position if the pattern matches.



T	9	6	4	5	5	6	5	0	0	1	8	9	0	8	9	7	7	6	5	0	0	1	1	2	3	5	4	8	9	6
P	6	5	0	0	1																									

- If the window is moved by one to the right, the new number can be computed in a simple way from the old number



$$64556 = (96455 - 9 \cdot 10^{m-1}) \cdot 10 + 6$$

Rabin-Karp Algorithm

Observations:

- In each step, we just have to compare two numbers.
- If the numbers are equal, the pattern appears at that position.
- When moving the window by one position, the new number can be computed from the old number in time $O(1)$.
- If we can compare two numbers in time $O(1)$, then the algorithm has a running time of $O(n)$.

- **Problem:** The numbers can be very large ($\Theta(m)$ bits)
  - Comparing two $\Theta(m)$-bit numbers requires time $\Theta(m)$
  - Not better than the naïve algorithm

- **Idea:** Apply hashing and compare hash values
  - If the window is moved by one to the right, we need to be able to compute the new hash value from the old hash value in time $O(1)$.

Rabin-Karp Algorithmus

Solution of Rabin and Karp:

- We calculate everything with numbers modulo M
  - M should be as large as possible, however still small enough such that numbers in the range $0, \dots, M-1$ fit in one memory cell (e.g., 64 Bit).
- Pattern and text window are then both numbers in

$$\{0, \dots, M-1\}$$

- When moving the search window, the new number can again be computed in time $O(1)$.
  - We will look at this afterwards...
- If the pattern is found, the two numbers are equal. If not, the can nevertheless be equal
  - If the numbers are equal, then we again check if we have found the pattern in a character-by-character way as in the naïve algorithm.

Rabin-Karp Algorithm: Example

Text: 572830354826 **Pattern: 283** **Modulus $M = 5$**

Pattern: $283 \bmod 5 = 3$

1st window: $572 \bmod 5 = 2$

2nd window: $728 \bmod 5 = 3$

in $O(1)$ Zeit

test: $728 \neq 283 \implies$ no match

3rd window: $283 \bmod 5 = 3$

test: $283 = 283 \implies$ pattern found

Computations Modulo M

$$x \bmod M = y \iff \exists q \in \mathbb{Z}\colon\ y = x + q \cdot M \wedge\ y \in \{0, \dots, M-1\}$$

- $x \bmod M$: add/subtract M from x until the result is in the range $\{0, \dots, M-1\}$

Some Rules:

$$(a \cdot b)\ \bmod M = \big((a \bmod M) \cdot (b \bmod M)\big) \bmod M$$

$$(a + b) \bmod M = \big((a \bmod M) + (b \bmod M)\big) \bmod M$$

$$\begin{aligned} a &= k \cdot M + c \implies a \bmod M = c \\ b &= \ell \cdot M + d \implies b \bmod M = d \end{aligned} \qquad (\, c, d \in \{0, \dots, M-1\}\,)$$

$$\begin{aligned} a \cdot b \bmod M &= \big(k\ell \cdot M^2 + (kd + \ell c) \cdot M + cd\big) \bmod M \\ &= cd \bmod M = (a \bmod M) \cdot (b \bmod M) \bmod M \end{aligned}$$

Computations Modulo M

$$x \bmod M = y \iff \exists q \in \mathbb{Z}\colon\ y = x + q \cdot M \ \land\ y \in \{0, \dots, M-1\}$$

- $x \bmod M$: add/subtract M from x until the result is in the range $\{0, \dots, M-1\}$

Some Rules:

$$(a \cdot b) \ \bmod M = \big((a \bmod M) \cdot (b \bmod M)\big) \bmod M$$
$$(a + b) \bmod M = \big((a \bmod M) + (b \bmod M)\big) \bmod M$$

Moving the Window:

- Moving window from position s to position $s+1$

$$t \coloneqq (T[s] \dots T[s+M-1]) \bmod M,$$
$$t' \coloneqq (T[s+1] \dots T[s+M]\,) \bmod M$$

$$t' = \Big(\big(t - T[s] \cdot \big(b^{M-1} \bmod M\big)\,\big) \cdot b + T[s+M]\Big) \bmod M$$

Computations Modulo M

$$x \bmod M = y \;\Leftrightarrow\; \exists q \in \mathbb{Z}\colon\; y = x + q \cdot M \;\wedge\; y \in \{0, \dots, M-1\}$$

Negative Numbers

- We need that $x \bmod M$ is always in $\{0, \dots, M-1\}$

  Examples:

  $$24 \bmod 10 = 4, \qquad 4 \bmod 10 = 4, \qquad -4 \bmod 10 = 6$$

- **But:** In Java / C++ / Python, we have $-x \,\%\, m = -(x \,\%\, m)$

  Examples:

  $$24 \,\%\, 10 = 4, \qquad 4 \,\%\, 10 = 4, \qquad -4 \,\%\, 10 = -4$$

- **Workaround:** If the result of $x \,\%\, M$ is negative, just add M to end up in the correct domain.

Rabin-Karp Algorithm: Pseudocode

Text $T[0 \ldots n-1]$, Pattern $P[0 \ldots m-1]$, Base b, Modulus M

```
$h = b^{m-1} \bmod M$
$p = 0;$  $t = 0;$
for $i = 0$ to $m-1$ do
    $p = (p \cdot b + P[i]) \bmod M$
    $t = (t \cdot b + T[i]) \bmod M$

for $s = 0$ to $n-m$ do
    if $p == t$ then
        TestPosition($s$)
    $t = \big((t - T[s] \cdot h) \cdot b + T[s+m]\big) \bmod M$
```

Can easily be computed in time $O(m)$ and if done right even in time $O(\log m)$

hash value of P: $p \coloneqq P \bmod M$

hash value of $T[0 \ldots m-1]$: $t \coloneqq T[0 \ldots m-1] \bmod M$

Time $O(m)$ if the hash values match

$h = b^{m-1} \bmod M$

update t in time $O(1)$

Rabin-Karp Algorithm: Running Time

Pre-Computation: $\boldsymbol{O(m)}$

In the worst case: $\boldsymbol{O(n \cdot m)}$

- The wirst case happens if the numbers match in each step. Then one has to check each of the m characters in each step to see if the pattern has really been found.
  - Should not happen too often is M is chosen in the right way...
  - Except if the pattern really occurs very often ($\Theta(n)$ times)...

In the best case: $\boldsymbol{O(n + k \cdot m)}$ (k: #occurrences of P in T)

- In the best case, the numbers are only equal if the pattern is really found. The time cost is then $O(n + k \cdot m)$, if the pattern appears k times in the text.

Choice of the Parameters …

Number Representation and Choice of M

- We would like that for $x \neq y$, it is "unlikely" that $h(x) = h(y)$ (for $h(x) \coloneqq x \bmod M$)
- We assume that the characters in pattern and text are represented as digits of a number in base-b representation
  - In our examples, we had $b = 10$
- If b and M have a common divisor, $h(x) = h(y)$ for $x \neq y$ is not so unlikely …

Extreme case $b = 10, M = 20$ (b is a divisor of M)

$$P = \alpha_{m-1}, \dots, \alpha_1, \alpha_0 = \sum_{i=0}^{m-1} \alpha_i \cdot 10^i \qquad 10^i \bmod 20 = \begin{cases} 1, & \text{if } i = 0 \\ 10, & \text{if } i = 1 \\ 0, & \text{if } i > 1 \end{cases}$$

$$\boldsymbol{P \bmod 20 = (\alpha_1 \cdot 10 + \alpha_0) \bmod 20}$$

Choice of the Parameters …

Number Representation and Choice of M

- We would like that for $x \neq y$, it is “unlikely” that $h(x) = h(y)$ (for $h(x) \coloneqq x \bmod M$)
- We assume that the characters in pattern and text are represented as digits of a number in base-b representation
  - In our examples, we had $b = 10$
- If b and M have a common divisor, $h(x) = h(y)$ for $x \neq y$ is not so unlikely …

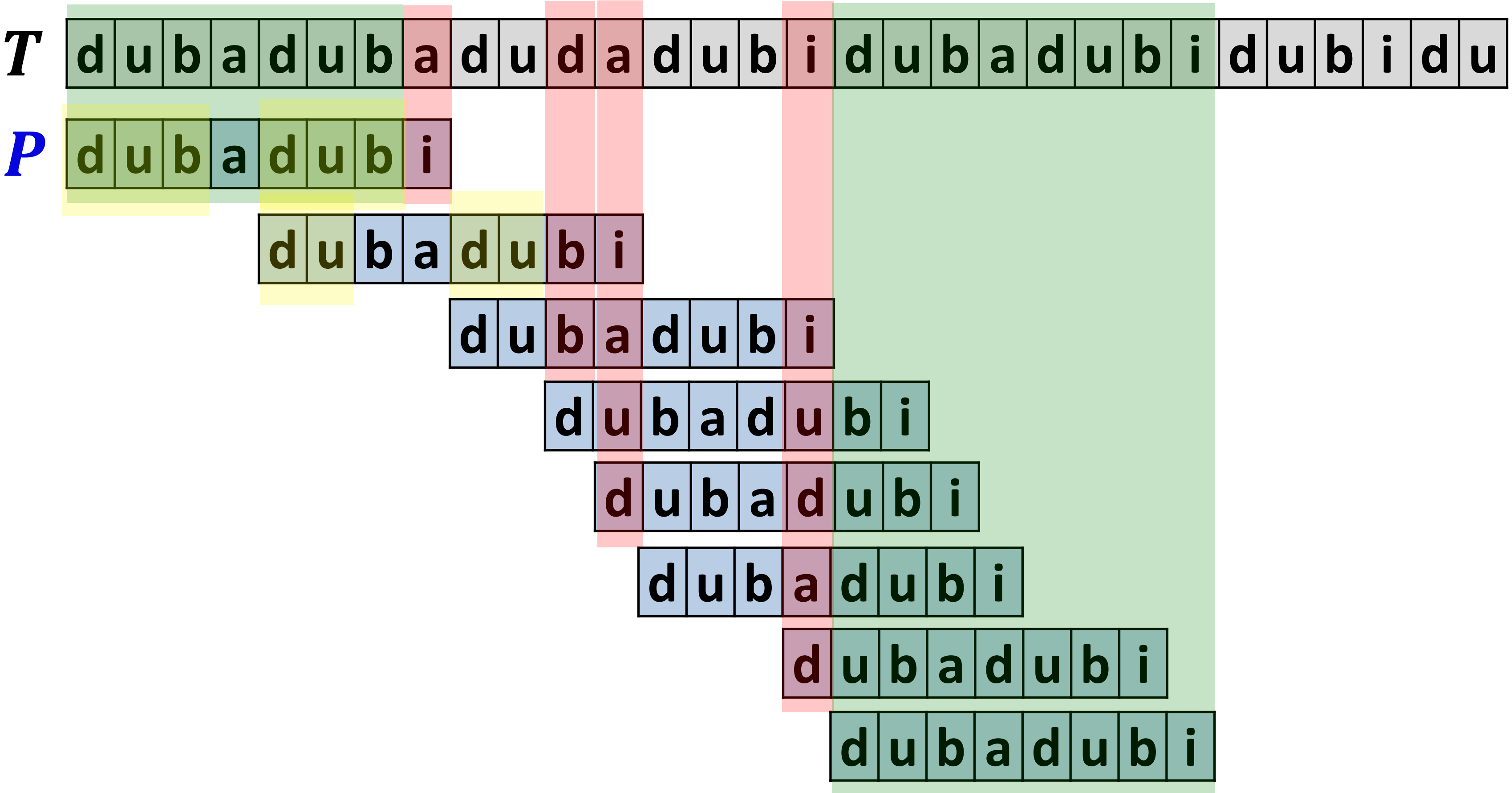
We therefore choose

- The base b as a sufficiently large prime number
  - For ASCII characters, we need $b > 256$
- M can then be chosen (almost) arbitrarily, ideally as a power of 2
  - Intermediate results are $< M \cdot b$, this should ideally fit within, e.g., 64 bits

Algorithm of Knuth, Morris, Pratt

- Can we always solve the problem in time $O(n)$?
  - in the worst case ...

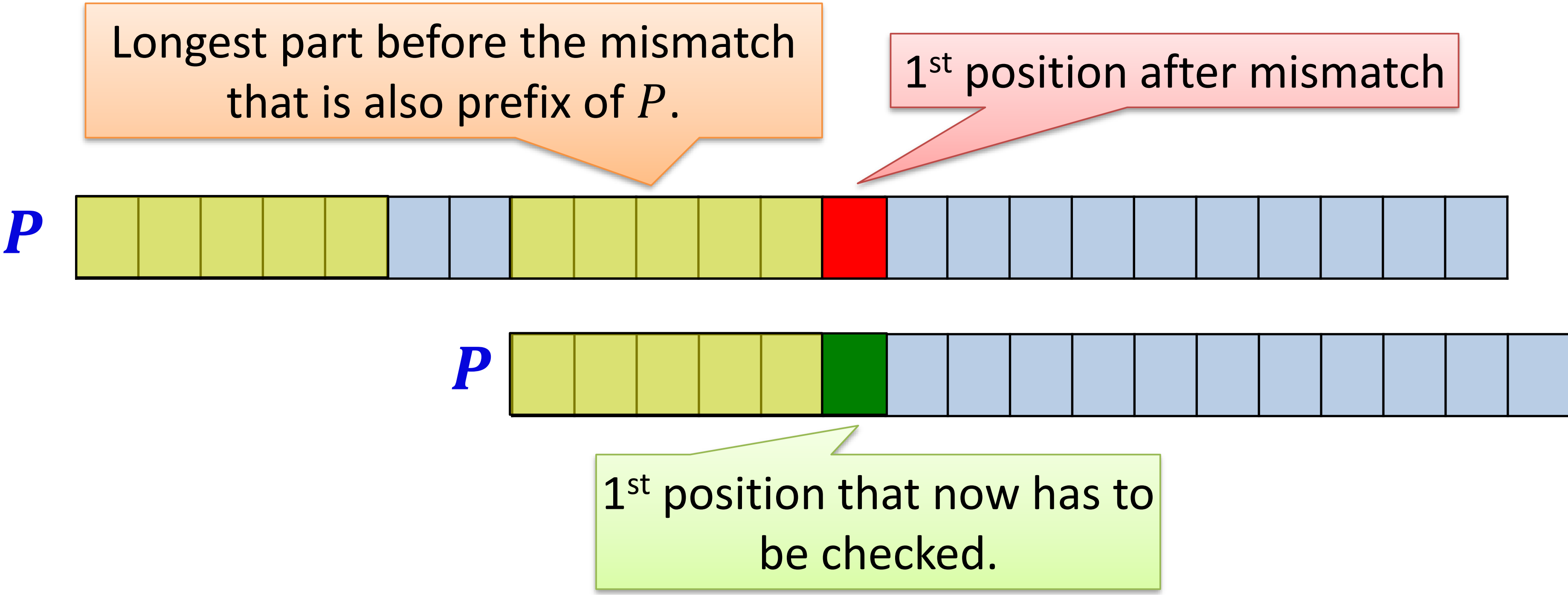
Let's again look at an example:



Knuth-Morris-Pratt Algorithm

Idea:

- If, when testing the pattern P at some position t we find that $P[t]$ does not match with the corresponding character in the text, then we know that the positions $P[0 \ldots t-1]$ were correct.
- This can be used in the remainder of the search


Longest part before the mismatch that is also prefix of P.

1^{st} position after mismatch

1^{st} position that now has to be checked.

Knuth-Morris-Pratt Algorithm

Precomputation: Array S of length $m + 1$

- $S[i]$: position in P, at which the search continues if when testing for the pattern, we have a mismatch at position i of the pattern
- $S[0] = -1, \quad S[1] = 0$
- $S[m]$: position in P, at which one continues after P has been found successfully.

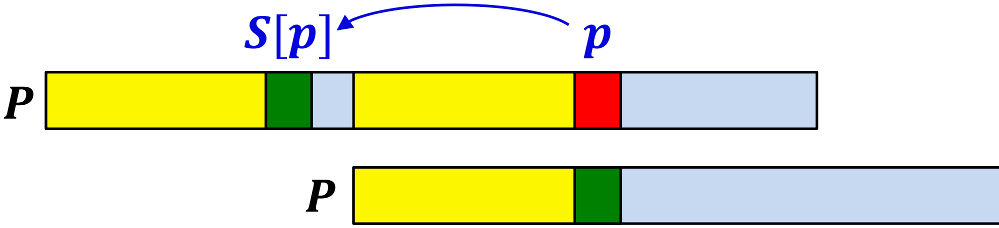
Example:

$P =$ `[ A, B, D, A, B, L, A, B, D, A, B, D]`

$S =$ `[-1, 0, 0, 0, 1, 2, 0, 1, 2, 3, 4, 5, 3]`

Knuth-Morris-Pratt Algorithm

```
$t = 0;\ p = 0$          // t: position in text,   p: position in pattern
while $t < n$ do
    if $T[t] == P[p]$ then                // characters match
        if $p == m - 1$ then              // pattern found
            pattern found at position $t - m + 1$
            $p = S[m];\ t = t + 1$
        else
            $p = p + 1;\ t = t + 1$
    else                                  // characters don't match
        if $p == 0$ then                  // mismatch at first character
            $t = t + 1$
        else
            $p = S[p]$
```



Knuth-Morris-Pratt Algorithm: Example

Pattern: ABCABC $S = [-1,0,0,0,1,2,3]$

Text:	A	D	A	B	C		D	A	B	C	A	G	A	B	V	A	B	C	A	B	C	A	B	C
	A	B	C	A	B	C																		
		A	B	C	A	B	C																	
			A	B	C	A	B	C																
						A	B	C	A	B	C													
							A	B	C	A	B	C												
								A	B	C	A	B	C											
											A	B	C	A	B	C								
												A	B	C	A	B	C							
													A	B	C	A	B	C						
															A	B	C	A	B	C				
																A	B	C	A	B	C			
																			A	B	C	A	B	C

Knuth-Morris-Pratt Alg.: Running Time

Running time without initialization of array S: $O(n)$

```
$t = 0;\ \ p = 0$
while $t < n$ do
    if $T[t] == P[p]$ then
        if $p == m - 1$ then
            pattern found
            $p = S[m];$ $t = t + 1$
        else
            $p = p + 1;$ $t = t + 1$
    else
        if $p == 0$ then
            $t = t + 1$
        else
            $p = S[p]$
```

In each step,

the position in the text is incremented

or

the window is moved

Initialization

Precomputation of Array S:

- $P =$ `[ A, B, D, A, B, L, A, B, D, A, B, D]`

  $S =$ `[-1, 0, 0, 0, 1, 2, 0, 1, 2, 3, 4, 5, 3]`

- At position i in S (for $i \in \{2, \dots, m\}$), we have

$$\boldsymbol{S[i] := \min_{k<i}\{P[i-k \dots i-1] = P[0 \dots k-1]\}}$$

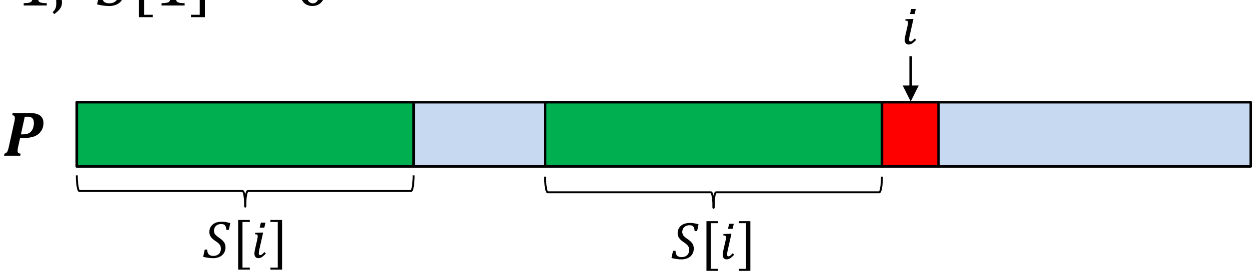
- $S[i]$: Length of the longest proper part of $P[0 \dots i-1]$, such that the part ends at position $i-1$ and the same part is also prefix of P.

Computation of $S[i]$:

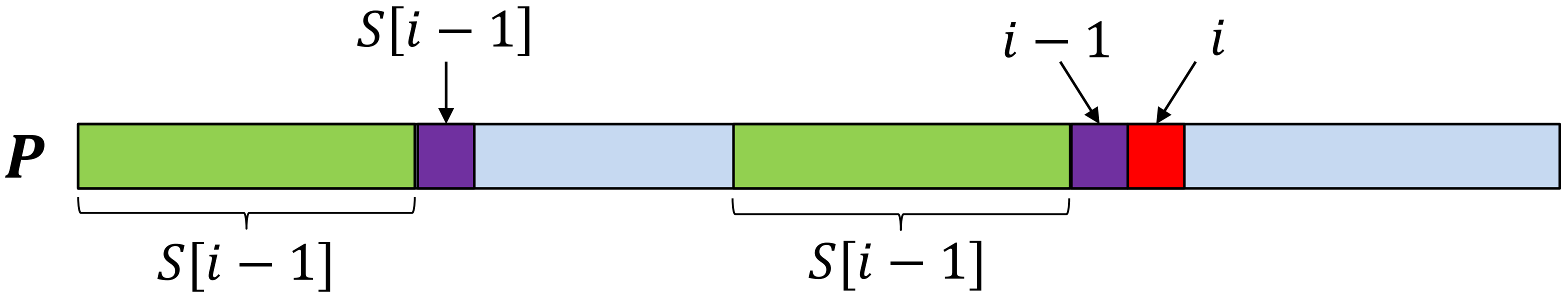
- We will look at this next…

Computation of S[i]

- $S[0] = -1,\ S[1] = 0$
- $i > 1$:



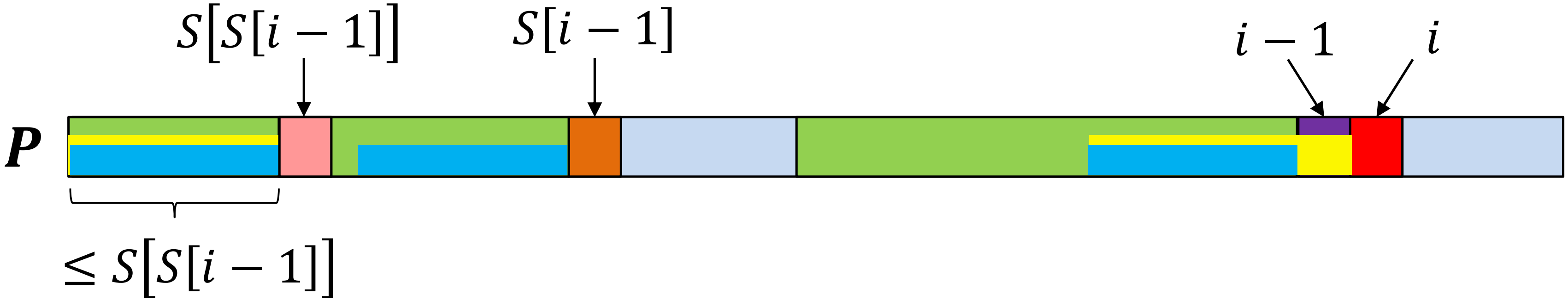
Case 1 : $\boldsymbol{P[i-1] = P\big[S[i-1]\big]}$



- If $P[i-1] = P\big[S[i-1]\big]$, then $S[i] = S[i-1] + 1$

Computation of S[i]

Case 2 : $\boldsymbol{P[i-1] \neq P\big[S[i-1]\big]}$



- Longest possible prefix and suffix has length $S\big[S[i-1]\big]+1$
  - Test if $P[i-1] = S\big[S[i-1]\big]$?
  - If yes, then we have $S[i] = S\big[S[i-1]\big]+1$
  - If no, then the next position we need to test is $S\Big[S\big[S[i-1]\big]\Big]$
  - etc.

Computation of $S[i]$: Pseudocode

```
$h = S[i-1]$
while $h \geq 0$ do
    if $P[i-1] == P[h]$ then
        $S[i] = h + 1$;  $h = -2$
    else
        $h = S[h]$
if $h == -1$ then $S[i] = 0$
```

Observation:

$S[i] \leq S[i-1] + 1$

If $S[i] = S[i-1] + 1$: 1 loop iteration

If $S[i] \leq S[i-1]$:

- Value of h decreases in each loop iteration
- At the end, we have $S[i] = h + 1$
- Number of loop iterations $\leq \Delta h + 1 = S[i-1] - S[i] + 2$

Computation of $S[i]$: Running Time

If $\boldsymbol{S[i] = S[i-1] + 1}$:

- #loop iterations $= 1 = S[i-1] - S[i] + 2$

Falls $\boldsymbol{S[i] \leq S[i-1]}$:

- #loop iterations $\leq \Delta h + 1 = S[i-1] - S[i] + 2$

Overall Running Time $\boldsymbol{T(m)}$:

$$\begin{aligned} T(m) &\leq \sum_{i=2}^{m} (S[i-1] - S[i] + 2) \\ &= 2(m-1) + (S[1] - S[2] + S[2] - S[3] + S[3] - \cdots \\ &\qquad\qquad + \ \ldots - S[m-1] + S[m-1] - S[m]) \\ &= 2(m-1) + S[1] - S[m] = O(m) \end{aligned}$$

Knuth-Morris-Pratt Algorithm: Summary

Knuth-Morris-Pratt Algorithm:

- First computes the array S of length $m + 1$ in time $O(m)$
  - only depends on the pattern P
  - describes at each position of the pattern, where (in the pattern) we have to continue after a mismatch
- With the help of S, all occurrences of the pattern P in the text T can be found in time $O(n)$.
  - In each step, one can either increment the current search position in the text T or one can move the position of the search window in T by at least 1 position to the right.

Overall Running Time: $\boldsymbol{O(m+n) = O(n)}$