Algorithm Design: Domain Specific Algorithms - Strings

Ștefan Ciobâcă, Dorel Lucanu

Faculty of Computer Science
Alexandru Ioan Cuza University, Iaand, România
dlucanu@info.uaic.ro

stefan.ciobaca@info.uaic.ro

PA 2019/2020

Plan

Alphabet, string

1. Alphabet = nonempty set A de characters
   $A_1 = \{A, a, B, b, C, c, \ldots, Z, z\}$
   $A_2 = \{0, 1\}$
   $A_3 = \mathbb{N}$
2. string of characters: a finite sequence of characters in A
   formal: $s : \{0, 1, \ldots, n-1\} \to A$
3. empty string: $\varepsilon : \emptyset \to A$
4. the length $|s|$ of a string s (or $length(s)$ or $s.length()$): the number of characters of the string
   $|\varepsilon| = 0$
   if $s : \{0, 1, \ldots, n-1\} \to A$ then $|s| = n$
5. $s[i]$ - character on position i $(0 \leq i \leq |s|)$;
6. substring: $s[i..j] = s[i]s[i+1]\ldots s[j]$ $(i \leq j)$

The set of strings A^*

- concatenation (product) of two stringsi s_1 and s_2: is the string s_1 immediately followed by s_2
  $s_1 s_2 = s_1[0] \dots s_1[|s_1| - 1] s_2[0] \dots s_2[|s_2| - 1]$
  $|s_1 s_2| = |s_1| + |s_2|$
- $s\varepsilon = \varepsilon s = s$
- A^* os the set of strings over alphabet A
  A^* the monoid freely generated by A
  if $A = \{a_1, \dots, a_m\}$, then $A^* = L((a_1 + \cdots a_m)^*)$ (we define later the regular expressions and their language)
- a language is a subset $L \subseteq A^*$

Factor, subsequence

- x is a factor of s iff there exists $u, v.$ s.t. $s = uxv$
  factor and *substring* are equivalent
  x is a proper factor iff $x \neq s$ (equivalently, $uv \neq \epsilon$)
- x is a prefix of s iff there exists $v.$ s.t. $s = xv$
  write $x \leq_{pref} s$
- x is a suffix of s if there exists $u.$ s.t. $s = ux$
  write $x \leq_{suff} s$
- x is subsequence of s iff there exists $|x| + 1$ strings $w_0, w_1, \ldots, w_{|x|}$ s.t. $s = w_0x[0]w_1x[1]\ldots x[|x| - 1]w_{|x|}$ (i.e., x is obtained by erasing $|y| - |x|$ characters in s); write $x \leq_{sseq} s$

Exercise

Are there $\leq_{pref}$ and $\leq_{suff}$ partial orders?

Lexicographic order

- lexicographic order:
  Assume a total order $\leq$ on A. Extend $\leq$ to $A^* \times A^*$ as follows:
  $s_1 \leq s_2$ iff $s_1 \leq_{pref} s_2$ or there exists $u, v, w \in A^*$ and $a, b \in A$ s.t. $s_1 = uav$, $s_2 = ubw$ and $a < b$.

Exercise. Write in Alk a function that decides the lexicographic order between two strings.

Occurrence

- x occurs in s if x is un factor of s
- x has an occurrence at start position i in s if $s[i] \ldots s[i + |x| - 1] = x$
- x has an occurrence at end position j in s if $s[j - |x| + 1] \ldots s[j] = x$
- first occurrence of x in s is the smallest start position (if there exists)

String Searching (Matching) Problem

Input Two strings: $s = s[0] \ldots s[n-1]$, called subject or text, and $p = p[0] \cdots p[m-1]$, called pattern.

Output The first occurrence of the pattern p in the text s, if any; -1, otherwise.

String Searching (Matching) Problem

Input Two strings: $s = s[0] \ldots s[n-1]$, called subject or text, and $p = p[0] \cdots p[m-1]$, called pattern.

Output The first occurrence of the pattern p in the text s, if any; -1, otherwise.

Variant: find all occurrences:

String Searching (Matching) Problem

Input Two strings: $s = s[0] \dots s[n-1]$, called subject or text, and $p = p[0] \cdots p[m-1]$, called pattern.

Output The first occurrence of the pattern p in the text s, if any; -1, otherwise.

Variant: find all occurrences:
Input Two strings: $s = s[0] \dots s[n-1]$, called subject or text, and $p = p[0] \cdots p[m-1]$, called pattern.

Output A set M consisting of all the occurrences of $p$in s.

The naive algorithm: helper

```
/*  @input: strings s[0..n-1], p[0..m-1],
            a position i, 0 <= i < n
    @output: true,  if p <=_pref s[i..n-1]
             false, otherwise  */
occAtPos(s, p,  i) {
  n = s.size();
  m = p.size();
  for (j = 0; j < m; ++j) {
    if (i + j >= n || s[i + j] != p[j]) {
      return false;
    }
  }
  return true;
}
```

The naive algorithm: helper

```
/*  @input: strings s[0..n-1], p[0..m-1],
            a position i, 0 <= i < n
    @output: true,  if p <=_pref s[i..n-1]
             false, otherwise  */
occAtPos(s, p,  i) {
  n = s.size();
  m = p.size();
  for (j = 0; j < m; ++j) {
    if (i + j >= n || s[i + j] != p[j]) {
      return false;
    }
  }
  return true;
}
```

Execution time: $O(m)$ in the worst case

The naive algorithm

```
/*
  @input:  strings s[0..n-1], p[0..m-1]
  @ouput:  the first occurence of p in s, if any
            -1, otherwise
*/
firstOcc(s, p)
{
  n = s.size();
  m = p.size();
  for (i = 0; i < n; ++i) {
    if (occAtPos(s, p, i)) {
      return i;
    }
  }
  return -1;
}
```

The naive algorithm: the worst case analysis

Time: $O(n \cdot m)$ in the worst case,

The naive algorithm: the worst case analysis

Time: $O(n \cdot m)$ in the worst case,

Exercise: 1. What is the worst case?
Remains the algorithm correct if we replace $i < n$ by $i < n - m$ in the loop `for`? What about the execution time?

The naive algorithm: expected time

Assume that A has d characters, $d \geq 2$.

$$X_{ij} = \begin{cases} 1 & , \text{s[i-1] and p[j-1] are compared} \\ 0 & , \textit{otherwise} \end{cases}$$

The naive algorithm: expected time

Assume that A has d characters, $d \geq 2$.

$$X_{ij} = \begin{cases} 1 & , \text{s[i-1] and p[j-1] are compared} \\ 0 & , \textit{otherwise} \end{cases}$$

$p_{ij} = Prob(X_{ij} = 1) = \frac{1}{d^{j-1}}$ (the first $j - 1$ characters in p must match)

The naive algorithm: expected time

Assume that A has d characters, $d \geq 2$.

$$X_{ij} = \begin{cases} 1 & \text{, s[i-1] and p[j-1] are compared} \\ 0 & \text{, } \mathit{otherwise} \end{cases}$$

$p_{ij} = Prob(X_{ij} = 1) = \dfrac{1}{d^{j-1}}$ (the first $j-1$ characters in p must match)
the number of comparisons $= \sum_{i=1}^{n+1-m} \sum_{j=1}^{m} X_{ij}$

The naive algorithm: expected time

Assume that A has d characters, $d \geq 2$.

$$X_{ij} = \begin{cases} 1 & \text{, s[i-1] and p[j-1] are compared} \\ 0 & , \textit{otherwise} \end{cases}$$

$p_{ij} = Prob(X_{ij} = 1) = \dfrac{1}{d^{j-1}}$ (the first $j - 1$ characters in p must match)
the number of comparisons $= \sum_{i=1}^{n+1-m} \sum_{j=1}^{m} X_{ij}$
the expected number of comparisons $= E(\sum_{i=1}^{n+1-m} \sum_{j=1}^{m} X_{ij})$

The naive algorithm: expected time

Assume that A has d characters, $d \geq 2$.

$$X_{ij} = \begin{cases} 1 & \text{, s[i-1] and p[j-1] are compared} \\ 0 & \text{, } \mathit{otherwise} \end{cases}$$

$p_{ij} = Prob(X_{ij} = 1) = \dfrac{1}{d^{j-1}}$ (the first $j - 1$ characters in p must match)
the number of comparisons $= \sum_{i=1}^{n+1-m} \sum_{j=1}^{m} X_{ij}$
the expected number of comparisons $= E(\sum_{i=1}^{n+1-m} \sum_{j=1}^{m} X_{ij})$
$\leq 2(n + 1 - m)$

Question: are there algorithms requiring $O(n)$ for the worst case?

A bit of history

- 1970, S.A. Cook: assumes that $\exists$ algorithms $O(n+m)$

A bit of history

- 1970, S.A. Cook: assumes that $\exists$ algorithms $O(n + m)$
- Knuth and Pratt au refined Cook's theory into an algorithm

A bit of history

- 1970, S.A. Cook: assumes that $\exists$ algorithms $O(n + m)$
- Knuth and Pratt au refined Cook's theory into an algorithm
- Morris independently discovered the same algorithm

A bit of history

- 1970, S.A. Cook: assumes that $\exists$ algorithms $O(n + m)$
- Knuth and Pratt au refined Cook's theory into an algorithm
- Morris independently discovered the same algorithm
- 1976: Knuth, Morris and Pratt algorithm (KMP)

A bit of history

- 1970, S.A. Cook: assumes that $\exists$ algorithms $O(n+m)$
- Knuth and Pratt au refined Cook's theory into an algorithm
- Morris independently discovered the same algorithm
- 1976: Knuth, Morris and Pratt algorithm (KMP)
- 1977: R.S. Boyer and J.S. Moore design an algorithm on a different idea

A bit of history

- 1970, S.A. Cook: assumes that $\exists$ algorithms $O(n + m)$
- Knuth and Pratt au refined Cook's theory into an algorithm
- Morris independently discovered the same algorithm
- 1976: Knuth, Morris and Pratt algorithm (KMP)
- 1977: R.S. Boyer and J.S. Moore design an algorithm on a different idea
- 1980: R.M. Karp and M.O. Rabin design an algorithm on a hashing-based idea (see the seminar)

Plan

Example

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
V	I	S	U	L		U	N	E	I		N	O	P	T	I		D	E
		$\neq$																
I	A	R																

19	20	21	22	23	24
	I	A	R	N	A

$\boxed{1}$

Example

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
V	I	S	U	L		U	N	E	I		N	O	P	T	I		D	E
					$\neq$													
			I	A	R													

19	20	21	22	23	24
	I	A	R	N	A

$\boxed{2}$

Example

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
V	I	S	U	L		U	N	E	I		N	O	P	T	I		D	E
								$\neq$										
						I	A	R										

19	20	21	22	23	24
	I	A	R	N	A

$\boxed{3}$

Example

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
V	I	S	U	L		U	N	E	I		N	O	P	T	I		D	E
											$\neq$							
									I	A	R							

19	20	21	22	23	24
	I	A	R	N	A

$\boxed{4}$

Example

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
V	I	S	U	L		U	N	E	I		N	O	P	T	I		D	E
														$\neq$				
												I	A	R				

19	20	21	22	23	24
	I	A	R	N	A

$\boxed{5}$

Example

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
V	I	S	U	L		U	N	E	I		N	O	P	T	I		D	E
																	$\neq$	
															I	A	R	

19	20	21	22	23	24
	I	A	R	N	A

$\boxed{6}$

Example

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
V	I	S	U	L		U	N	E	I		N	O	P	T	I		D	E
																		I

19	20	21	22	23	24
	I	A	R	N	A
	$\neq$				
A	R				

$\boxed{7}$

Example

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
V	I	S	U	L		U	N	E	I		N	O	P	T	I		D	E

19	20	21	22	23	24
	I	A	R	N	A
			=		
	I	A	R		

$\boxed{8}$

Example

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
V	I	S	U	L		U	N	E	I		N	O	P	T	I		D	E

19	20	21	22	23	24
	I	A	R	N	A
		=			
	I	A	R		

$\boxed{9}$

Example

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
V	I	S	U	L		U	N	E	I		N	O	P	T	I		D	E

19	20	21	22	23	24
	I	A	R	N	A
	=				
	I	A	R		

$\boxed{10}$

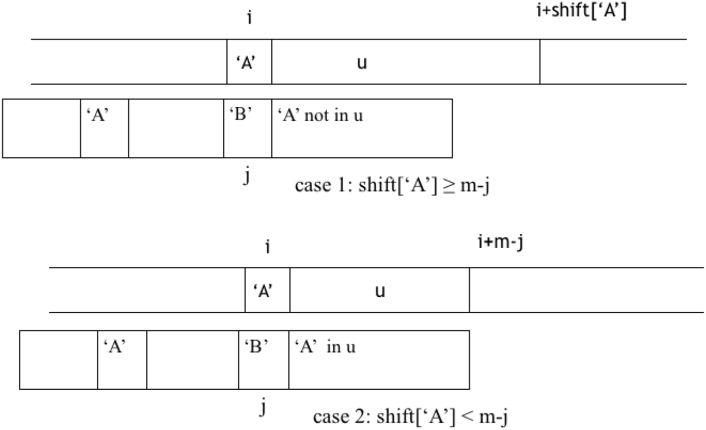
Bad character shift rule 1/3

Avoids unsuccessful comparisons with characters from the subject that do not occur in the pattern or in a (maximal) suffix of it.

$$
\mathit{shift}[C] = \begin{cases} (m-1) - \text{the last occurrence position of } C \text{ in pattern} & \text{, if } C \text{ occurs in pattern} \\ m & \text{, otherwise} \end{cases}
$$

(alternatively, $\mathit{shift}(C) = \max(\{0\} \cup \{i < m \mid p[i] = C\})$)

Bad character shift rule 2/3



First case: `i = i + shift[s[i]];`
Second case: `i = i + m - j;`

Bad character shift rule 3/3

If $p[j] \neq s[i] = C$,

1. if the rightmost occurrence of C in p is $k < j$, $p[k]$ and $s[i]$ will be aligned (i = i + shift[s[i]])
2. if the rightmost occurrenceof C in p is $k > j$, p is sfifted to right with one position (i = i + m - j)
3. if C does not occur in p, the pattern p is aligned with $s[i+1..i+m]$ (i = i + m). Become a particular case of the first one if $shift[s[i]] = m$.

Boyer-Moore Algorithm (version 1)

```
BM(s, p, shift) {
  n = s.size();
  m = p.size();
  i = m-1; j = m-1;
  repeat
    if (s[i] == p[j]) {
      i = i-1;
      j = j-1;
    }
    else {
      if ((m-j) > shift[s[i]]) i = i+m-j;
      else i = i+shift[s[i]];
      j = m-1;
    }
  until (j<0 or i>n-1);
  if (j<0)  return i+1;
  else return -1;
}
```

Analysis

Worst case: $O(m \cdot n)$.

Expected time is much better.

We will see later that.

Plan

Motivation

Bad character shift rule is inefficient if the alphabet is small (binary, for example).

In such cases the algorithm could be more efficient if uses the information gained from compared suffixes.

This case is called the good suffix rule.

The good suffix rule: case 1

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16		17	18	19	20	21	22	23	24
a	c	c	a	c	b	b	c	a	b	b	a	c	b	a	b	a	b	b	a	b	b	c	b	a	b

a	a	b	a	b	b	c	b	a	b
0	1	2	3	4	5	6	7	8	9

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16		17	18	19	20	21	22	23	24
a	c	c	a	c	b	b	c	a	b	b	a	c	b	a	b	a	b	b	a	b	b	c	b	a	b

a	a	b	a	b	b	c	b	a	b
0	1	2	3	4	5	6	7	8	9

$i-1 = j-1 = 7$, $m = 10$, $s[i-1] \neq p[j-1]$, $p[j..m-1] = s[i..i+m-j-1]$
$p[1..2] = p[8..9]$, $p[0] \neq p[7]$ and $p[1..2]$ is closest to $p[8..9]$ having this property

The good suffix rule: case 1, formally

Case 1:
if $p[j-1]$ does not match and p includes a copy of $p[j..m-1]$ preceded by a character $\neq p[j-1]$, then shift to the closest copy from left with this property.

The good suffix rule: case 2

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16		17	18	19	20	21	22	23	24
a	b	c	a	c	b	b	c	a	b	b	a	c	b	a	b	a	b	b	a	b	b	c	b	a	b

a	b	b	a	b	c	b	c	a	b
0	1	2	3	4	5	6	7	8	9

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16		17	18	19	20	21	22	23	24
a	b	c	a	c	b	b	c	a	b	b	a	c	b	a	b	a	b	b	a	b	b	c	b	a	b

a	b	b	a	b	c	b	c	a	b
0	1	2	3	4	5	6	7	8	9

$i-1 = j-1 = 5$, $m = 10$, $p[j..m-1] = s[i+m-j-1]$,
$p[0..1] = s[8..9]$ is the longest prefix of p that is a suffix of $s[0..9]$

The good suffix rule: case 2, formally

Case 2:

– if Case 1 is not applicable, then do the smallest shift such that the suffix of $s[0..i{+}m{-}j{-}1]$ is matched by a prefix of p

– if longest suffix of $s[0..i{+}m{-}j{-}1]$ matched by a prefix of p is the empty string, then shift with m positions

– but "suffix of $s[0..i{+}m{-}j{-}1]$ is matched by a prefix of p" is equivalent to "a suffix of p is matched by a prefix of p

– proper factor that is prefix and suffix is called border

$goodSuff(j)$ - definition (Case 1)

$goodSuff(j) =$ the end position of the occurrence of $p[j..m-1]$ closest to j and and it is not preceded by $p[j-1]$.

If such a copy does not exists, $goodSuff(j) = 0$.

We have $0 \leq goodSuff(j) < m - 1$.

Proposition

The values of $goodSuff(j)$ can be computed inl $O(m)$ time.

The proof at seminar.

Preprocessing in Case 2

$lp(j)$ = the length of the longest prefix of p that is suffix of $p[j..m-1]$.

We have

Proposition

$lp(j)$ can be computed in $O(m)$ time.

The proof at seminar.

Good Suffix Rule

Assume that $p[j-1]$ does not match (aftert $p[j..m-1]$ matched).

1. if $goodSuff(j) > 0$,the shift with $m - goodSuff(j)$ positions (case 1)
2. if $goodSuff(j) = 0$, the shift with $m - lp(j)$ (case 2)

If $p[m-1]$ matches, then $j = m$ and the shift is correct.

Boyer-Moore Algorithm (version 2)

```
BM(s, n, p, m, goodSuff, lp) {
  k = m-1;
  while (k < n) {
    i = k;  j = m-1;
    while (j > 0 && p[j] == s[i]) {
      i = i-1;
      j = j-1;
    }
    if (j < 0) return i+1;
    otherwise p[j] does not match and  shift with the maximum of the
  values returned bu the bad character shift rule and the goof suffix rule
  }
}
```

Boyer-Moore Algorithm: summary

- $O(n+m)$ time if the pattern p does not occur in the text; otherwise it remains $O(m \cdot n)$
- however, with a simple modification (Galil rule, 1979) $O(n+m)$ time can be obtained in all the cases
- the original algorithm of Boyer-Moore (1977) uses a simplified form of the good suffix rule
- the first proof for $O(n+m)$, when the pattern p does not occur in the text, was given by Knuth, Morris and Pratt (1977); a different proof was independently given by Guibas and Odlyzko (1980)
- Richard Colen (1991) established a limit of $4n$ (with a easier proof), then a limit of $3n$ (with a more complex proof)

Plan

Naive Algorithm[1]

a	b	c	b	a	b	a	b	a	a	b	c	b	a	b
				a	b	a	b	a	c	a				

[1] Exemplu din [CLRS]

Naive Algorithm[1]

a	b	c	b	a	b	a	b	a	a	b	c	b	a	b
				$=$										
				a	b	a	b	a	c	a				

[1] Exemplu din [CLRS]

Naive Algorithm[1]

a	b	c	b	a	b	a	b	a	a	b	c	b	a	b
				$=$	$=$									
				a	b	a	b	a	c	a				

[1] Exemplu din [CLRS]

Naive Algorithm[1]

a	b	c	b	a	b	a	b	a	a	b	c	b	a	b
				$=$	$=$	$=$								
				a	b	a	b	a	c	a				

[1] Exemplu din [CLRS]

Naive Algorithm[1]

a	b	c	b	a	b	a	b	a	a	b	c	b	a	b
				$=$	$=$	$=$	$=$							
				a	b	a	b	a	c	a				

[1]Exemplu din [CLRS]

Naive Algorithm[1]

a	b	c	b	a	b	a	b	a	a	b	c	b	a	b
				$=$	$=$	$=$	$=$	$=$						
				a	b	a	b	a	c	a				

[1] Exemplu din [CLRS]

Naive Algorithm[1]

a	b	c	b	a	b	a	b	a	a	b	c	b	a	b
				$=$	$=$	$=$	$=$	$=$	$\neq$					
				a	b	a	b	a	c	a				

[1] Exemplu din [CLRS]

Intuition[2]

a	b	c	b	a	b	a	b	a	a	b	c	b	a	b
				$=$	$=$	$=$	$=$	$=$	$\neq$					
				a	b	a	b	a	c	a				
					a	b	a	b	a	c	a			
						a	b	a	b	a	c	a		

[2] Exemplu din [CLRS]

Intuition[3]

?	?	?	?	a	b	a	b	a	?	?	?	?	?	?
				$=$	$=$	$=$	$=$	$=$	$\neq$					
				a	b	a	b	a	?	?				
					a	b	a	b	?	?	?			
						a	b	a	?	?	?	?		

[3] Exemplu din [CLRS]

Intuition[3]

?	?	?	?	a	b	a	b	a	?	?	?	?	?	?
				$=$	$=$	$=$	$=$	$=$	$\neq$					
				a	b	a	b	a	?	?				
					a	b	a	b	?	?	?			
						a	b	a	?	?	?	?		

For the pattern *ababaca*, if at position i exact the first 5 characters match, then there is no chance that the pattern match at position $i+1$. But we have chance at $i+2$. Why?

[3] Exemplu din [CLRS]

Ideea

?	?	?	?	x_0	x_1	x_2	x_3	x_4	x_5	x_6	?	?	?	?	?	?	?	?
				$=$	$=$	$=$	$=$	$=$	$=$	$=$	$\neq$							
				x_0	x_1	x_2	x_3	x_4	x_5	x_6	?							
								$=$	$=$	$=$								
								x_0	x_1	x_2	x_3	x_4	x_5	x_6	?			

Idea

?	?	?	?	x_0	x_1	x_2	x_3	x_0	x_1	x_2	?	?	?	?	?	?	?	?
				$=$	$=$	$=$	$=$	$=$	$=$	$=$	$\neq$							
				x_0	x_1	x_2	x_3	x_0	x_1	x_2	?							
								$=$	$=$	$=$								
								x_0	x_1	x_2	x_3	x_0	x_1	x_2	?			

Idea

?	?	?	?	x_0	$\ldots$	x_{k-1}	$\ldots$	x_0	$\ldots$	x_{k-1}	?	?	?	?
				$=$	$=$	$=$	$=$	$=$	$=$	$=$	$\neq$			
				x_0	$\ldots$	x_{k-1}	$\ldots$	x_0	$\ldots$	x_{k-1}	?			

Idea

?	?	?	?	x_0	$\dots$	x_{k-1}	$\dots$	x_0	$\dots$	x_{k-1}	?	?	?	?
				$=$	$=$	$=$	$=$	$=$	$=$	$=$	$\neq$			
				x_0	$\dots$	x_{k-1}	$\dots$	x_0	$\dots$	x_{k-1}	?			

We are interested to find the largest k s.t. $x_1 \dots x_k$ is both prefix and suffix of the matched prefix.

Notations

- reminder: border (frontier) of a string t - un factor that is both prefix and suffix of t
- notation: $maxFr(k)$ - the maximum border of $p[0..k-1]$ that is proper factor of ($\neq p[0..k-1]$)
  $f[k] = |maxFr(k)|$ (the length of the longest border of $p[0..k-1]$)
- example: $p = \begin{array}{ccccccc} a & b & a & b & a & c & a \\ 0 & 1 & 2 & 3 & 4 & 5 & 6 \end{array}$

k	$maxFr(k)$	$f[i]$
1	ε	0
2	ε	0
3	a	1
4	ab	2
5	aba	3
6	ε	0

Notations

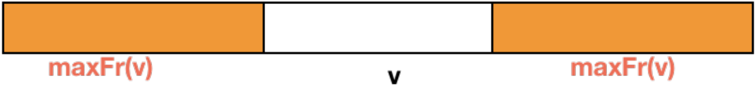
- reminder: border (frontier) of a string t - un factor that is both prefix and suffix of t
- notation: $maxFr(k)$ - the maximum border of $p[0..k-1]$ that is proper factor of ($\neq p[0..k-1]$)
  $f[k] = |maxFr(k)|$ (the length of the longest border of $p[0..k-1]$)
- example: $p = \begin{array}{ccccccc} a & b & a & b & a & c & a \\ 0 & 1 & 2 & 3 & 4 & 5 & 6 \end{array}$

k	$maxFr(k)$	$f[i]$
1	ε	0
2	ε	0
3	a	1
4	ab	2
5	aba	3
6	ε	0

- notation: $u \leq_{fr} v$ iff $u \leq_{pref} v$ and $u \leq_{suff} v$

Reasoning problem domain 1/4

- formal definition of of $maxFr(v)$:
  $maxFr(v) <_{fr} v$



and

$(\forall w) w <_{fr} v$ implies $w \leq_{fr} maxFr(v)$;
i.e., the maximum border is maximum relative to $\leq_{fr}$ as well.

Reasoning problem domain 2/4

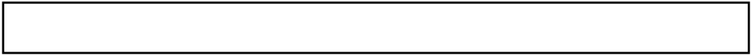
- notation: $maxFr^0(v) = v$, $maxFr^{j+1}(v) = maxFr(maxFr^j(v))$

Reasoning problem domain 2/4

- notation: $maxFr^0(v) = v$, $maxFr^{j+1}(v) = maxFr(maxFr^j(v))$
- we have:
  $\cdots <_{fr} maxFr^{j+1}(v) <_{fr} maxFr^j(v) <_{fr} \cdots <_{fr} maxFr^1(v) <_{fr} maxFr^0(v) = v$

Reasoning problem domain 2/4

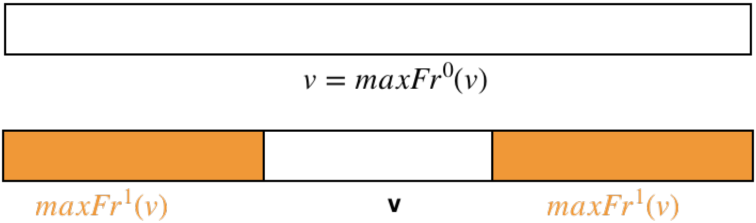
- notation: $maxFr^0(v) = v$, $maxFr^{j+1}(v) = maxFr(maxFr^j(v))$
- we have:
  $\cdots <_{fr} maxFr^{j+1}(v) <_{fr} maxFr^j(v) <_{fr} \cdots <_{fr} maxFr^1(v) <_{fr} maxFr^0(v) = v$



$$v = maxFr^0(v)$$

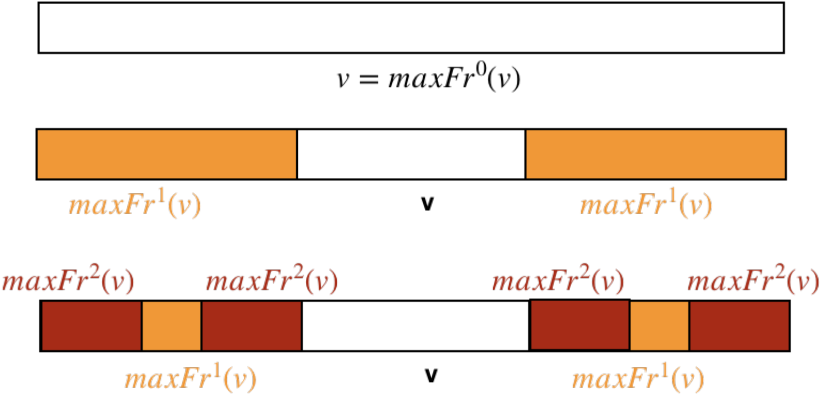
Reasoning problem domain 2/4

- notation: $maxFr^0(v) = v$, $maxFr^{j+1}(v) = maxFr(maxFr^j(v))$
- we have:
  $\cdots <_{fr} maxFr^{j+1}(v) <_{fr} maxFr^j(v) <_{fr} \cdots <_{fr} maxFr^1(v) <_{fr} maxFr^0(v) = v$



Reasoning problem domain 2/4

- notation: $maxFr^0(v) = v$, $maxFr^{j+1}(v) = maxFr(maxFr^j(v))$
- we have:
  $\cdots <_{fr} maxFr^{j+1}(v) <_{fr} maxFr^j(v) <_{fr} \cdots <_{fr} maxFr^1(v) <_{fr} maxFr^0(v) = v$



Reasoning problem domain 3/4

Theorem

$u \leq_{fr} v$ dif there exists $j \geq 0$ s.t. $u = maxFr^{j}(v)$.

Reasoning problem domain 3/4

Theorem

$u \leq_{fr} v$ dif there exists $j \geq 0$ s.t. $u = maxFr^j(v)$.

Corollary

$u <_{fr} v$ if there exists $j > 0$ s.t. $u = maxFr^j(v)$.

Reasoning problem domain 4/4

Theorem

$|maxFr^j(p[0..k-1])| = f^j[k]$.

Since the borders ofi $v = p[0..k-1]$ are

$$\cdots <_{fr} maxFr^{j+1}(v) <_{fr} maxFr^j(v) <_{fr} \cdots <_{fr} maxFr^1(v) <_{fr} maxFr^0(v) = v$$

it follows that their lengths satisfy the relation

$$\cdots < f^{j+1}[k] < f^j[k] < \cdots < f[k] < f^0[k] = k$$

and the "shift" from $maxFr^{j+1}(v)$ to $maxFr^j(v)$ is equal to $f^{j+1}[k] - f^j[k]$

Example 1/6

Here is an example how $f[i]$ is used for an efficient searching:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
a	b	c	b	a	b	a	b	a	b	a	b	a	c	a
$=$	$=$	$\neq$												
a	b	a	b	a	c	a								
0	1	2	3	4	5	6								

$f =$

0	1	2	3	4	5	6
a	b	a	b	a	c	a
-1	0	0	1	2	3	0

- failure at position $i = k = 2$
- $f[k] = f[2] = 0$
- shift with $k - f[k] = 2 - 0 = 2$ positions
- the next position to be compared: $i = 2, k = f[k] = 0$

Example 2/6

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
a	b	c	b	a	b	a	b	a	b	a	b	a	c	a
		$\neq$												
		a	b	a	b	a	c	a						
		0	1	2	3	4	5	6						

$f =$

0	1	2	3	4	5	6
a	b	a	b	a	c	a
-1	0	0	1	2	3	0

- failure at $i = 2, k = 0$
- $f[0] =?$
- shift $k - f[k] = 1$, so $f[0] = -1$
- the next position to be compared: $i = i + 1 = 3, k = f[k] + 1 = 0$

Example 3/6

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
a	b	c	b	a	b	a	b	a	b	a	b	a	c	a
			$\neq$											
			a	b	a	b	a	c	a					
			0	1	2	3	4	5	6					

$f =$

0	1	2	3	4	5	6
a	b	a	b	a	c	a
-1	0	0	1	2	3	0

- failure at $i = 3, k = 0$
- $f[0] = -1$
- shift with $k - f[k] = 0 - f[0] = 1$ positions
- the next position to be compared: $i = i + 1 = 4, k = f[k] + 1 = 0$

Example 4/6

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
a	b	c	b	a	b	a	b	a	b	a	b	a	c	a
				$=$	$=$	$=$	$=$	$=$	$\neq$					
				a	b	a	b	a	c	a				
				0	1	2	3	4	5	6				

$f =$

0	1	2	3	4	5	6
a	b	a	b	a	c	a
-1	0	0	1	2	3	0

- failure at $i = 9, k = 5$
- $f[5] = 3$
- shift $k - f[k] = 5 - f[5] = 2$ positions
- the next position to be compared: $i = 9, k = f[k] = 3$

Example 5/6

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
a	b	c	b	a	b	a	b	a	b	a	b	a	c	a
									$=$	$=$	$\neq$			
						a	b	a	b	a	c	a		
						0	1	2	3	4	5	6		

$f =$

0	1	2	3	4	5	6
a	b	a	b	a	c	a
-1	0	0	1	2	3	0

- failure at $i = 11, k = 5$
- $f[5] = 3$
- shift $k - f[k] = 5 - f[5] = 2$ positions
- the next position to be compared: $i = 11, k = f[k] = 3$

Example 6/6

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
a	b	c	b	a	b	a	b	a	b	a	b	a	c	a
											$=$	$=$	$=$	$=$
								a	b	a	b	a	c	a
								0	1	2	3	4	5	6

- the first occurrence of the pattern matching

Rules

- if $p[k] == s[i]$ and $k = m - 1$, then we have an occurrence of the patterns at the start position $i - m + 1$;
- if $p[k] \neq s[i]$ then k becomes $f[k]$ ($k = f[k]$), i.e., we test the next border;
- if $k == -1$ then increment both i and k;
- if $p[k] == s[i]$ and $k < m - 1$ then increment botht i and k;

KMP Algorithm in Alk

```
KMP(s, p, f) {
  n = s.size();
  m = p.size();
  i = 0;
  k = 0;
  while (i < n) {
    while ((k != -1) && (p[k] != s[i]))
      k = f[k];
    // k == -1 or p[k] == s[i]
    if (k == m-1)
      return i-m+1; /* gasit p in s */
    else {
      i = i+1;
      k = k+1;
    }
  }
  return -1; /* p nu apare in s */
}
```

Execution Time

1. Remarks:
   1. for any k, $-1 \leq f[k] < k$.

Execution Time

1. Remarks:
   1. for any k, $-1 \leq f[k] < k$.
   2. the value k increases at most n times (the same is true for i)

Execution Time

1. Remarks:
   1. for any k, $-1 \leq f[k] < k$.
   2. the value k increases at most n times (the same is true for i)
   3. at each inner while loop k decreases, but it is always ≥ -1

Execution Time

1. Remarks:
    1. for any k, $-1 \leq f[k] < k$.
    2. the value k increases at most n times (the same is true for i)
    3. at each inner while loop k decreases, but it is always ≥ -1
    4. per total, k cannot decrease more times than it increases

Execution Time

1. Remarks:
   1. for any k, $-1 \leq f[k] < k$.
   2. the value k increases at most n times (the same is true for i)
   3. at each inner while loop k decreases, but it is always ≥ -1
   4. per total, k cannot decrease more times than it increases
   5. so the inner while will execute at most n iterations in total

Execution Time

1. Remarks:
   1. for any k, $-1 \leq f[k] < k$.
   2. the value k increases at most n times (the same is true for i)
   3. at each inner while loop k decreases, but it is always ≥ -1
   4. per total, k cannot decrease more times than it increases
   5. so the inner while will execute at most n iterations in total
2. Conclusion: the execution time for KMP is $O(n)$

Failure function f: introduction

- since f is used when a comparison fails, f is also called failure function
- usualy denoted by π (e.g., in [CLR])
- recall that $f[i] = |maxFr(p[0..i-1])|$ (the length of the maximum border of $p[0..i-1]$
- example:

a	b	a	b	a	c	a
-1	0	0	1	2	3	0

Failure function f: introduction

- since f is used when a comparison fails, f is also called failure function
- usualy denoted by π (e.g., in [CLR])
- recall that $f[i] = |maxFr(p[0..i-1])|$ (the length of the maximum border of $p[0..i-1]$
- example:

a	b	a	b	a	c	a
-1	0	0	1	2	3	0

- A naive implementation with $O(m^3)$ time is possible (exercise)

Failure function f: introduction

- since f is used when a comparison fails, f is also called failure function
- usualy denoted by π (e.g., in [CLR])
- recall that $f[i] = |maxFr(p[0..i-1])|$ (the length of the maximum border of $p[0..i-1]$
- example:

a	b	a	b	a	c	a
-1	0	0	1	2	3	0

- A naive implementation with $O(m^3)$ time is possible (exercise)
- Question: if $f[0..i-1]$ is already computed, how $f[i]$ can be efficiently computed?

Failure function f: computation

- recall that the borders of $v = p[0..i-1]$ are:
  $\cdots <_{fr} maxFr^{j+1}(v) <_{fr} maxFr^{j}(v) <_{fr} \cdots <_{fr} maxFr^{1}(v) <_{fr} maxFr^{0}(v) = v$

  and $f^j[i] = |maxFr^j(p[0..i-1])$

Failure function f: computation

- recall that the borders of $v = p[0..i-1]$ are:
  $\cdots <_{fr} maxFr^{j+1}(v) <_{fr} maxFr^{j}(v) <_{fr} \cdots <_{fr} maxFr^{1}(v) <_{fr} maxFr^{0}(v) = v$
  and $f^j[i] = |maxFr^j(p[0..i-1])$
- it follows that $f[i] = f^k[i-1] + 1$, where k is the smallest integer having the property $p[f^k[i-1]+1] = p[i-1]$

Failure function f: computation

- recall that the borders of $v = p[0..i-1]$ are:
  $\cdots <_{fr} maxFr^{j+1}(v) <_{fr} maxFr^{j}(v) <_{fr} \cdots <_{fr} maxFr^{1}(v) <_{fr} maxFr^{0}(v) = v$
  and $f^j[i] = |maxFr^j(p[0..i-1])$
- it follows that $f[i] = f^k[i-1] + 1$, where k is the smallest integer having the property $p[f^k[i-1]+1] = p[i-1]$
- i.e., we look at the prefixes of p that are suffixes of $p[0..i-2]$ and take the largest for that the next character is equal to $p[i-1]$

Failure function f: computation

- recall that the borders of $v = p[0..i-1]$ are:
  $\cdots <_{fr} maxFr^{j+1}(v) <_{fr} maxFr^{j}(v) <_{fr} \cdots <_{fr} maxFr^{1}(v) <_{fr} maxFr^{0}(v) = v$
  and $f^j[i] = |maxFr^j(p[0..i-1])$
- it follows that $f[i] = f^k[i-1]+1$, where k is the smallest integer having the property $p[f^k[i-1]+1] = p[i-1]$
- i.e., we look at the prefixes of p that are suffixes of $p[0..i-2]$ and take the largest for that the next character is equal to $p[i-1]$
- but the values $f^j[i-1]$, $j = 0, 1, \ldots$ are in $f[0..i-1]$! (which is already computed)

Failure function f: Alk description

```
f[0] = -1; f[1] = 0;
k = 0;
for (i = 2; i < m; ++i) {
  // invariant:  k = f[i-1]
  while(k >= 0 && p[k] != p[i-1])
    // invariant: there exists j cu k = f^j[i-1] si
    // j is cel mai mic cu p[f^j[i-1]+1] != p[i-1]
    k = f[k];
  k = k + 1;
  f[i] = k;
}
```

Failure function f: Alk description

```
f[0] = -1; f[1] = 0;
k = 0;
for (i = 2; i < m; ++i) {
  // invariant:  k = f[i-1]
  while(k >= 0 && p[k] != p[i-1])
    // invariant: there exists j cu k = f^j[i-1] si
    // j is cel mai mic cu p[f^j[i-1]+1] != p[i-1]
    k = f[k];
  k = k + 1;
  f[i] = k;
}
```

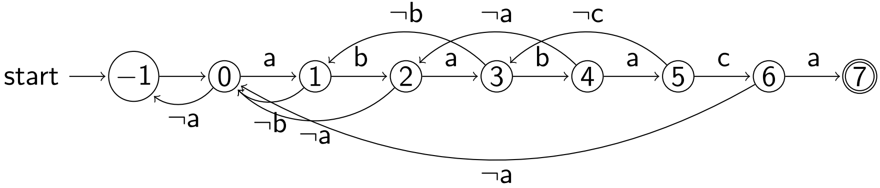
Execution time: $\Theta(m)$.

Failure function f: Alk description

```
f[0] = -1; f[1] = 0;
k = 0;
for (i = 2; i < m; ++i) {
  // invariant:  k = f[i-1]
  while(k >= 0 && p[k] != p[i-1])
    // invariant: there exists j cu k = f^j[i-1] si
    // j is cel mai mic cu p[f^j[i-1]+1] != p[i-1]
    k = f[k];
  k = k + 1;
  f[i] = k;
}
```

Execution time: $\Theta(m)$.
The analysis is similar to that of KMP.

Failure function as an automaton



An automaton consists of:

- input alphabet (e.g, $a, b, c, \ldots$)
- state (e.g., $-1, 0, 1, \ldots, 7$)
- initial state (-1 in the example)
- accepting (final) state (7 in the example)
- instantaneous transitions: (e.g., $-1 \to 0$)
- labeled transitions: $(0 \xrightarrow{a} 1, 1 \xrightarrow{b} 2, 2 \xrightarrow{a} 3, \ldots, 0 \to -1, 1 \to 0, \ldots)$

Plan

Motivation: patterns in many text editors (e.g., Emacs)

From documentation (Emacs):

Pattern	Matches
.	Any single character except newline ("\n").
\.	One period
[0-9]+	One or more digits
[^ 0-9]+	One or more non-digit characters
[A-Za-z]+	one or more letters
[-A-Za-z0-9]+	one or more letter, digit, hyphen
[_A-Za-z0-9]+	one or more letter, digit, underscore
[-_A-Za-z0-9]+	one or more letter, digit, hyphen, underscore
[[:ascii:]]+	one or more ASCII chars. (codepoint 0 to 127, inclusive)
[[:nonascii:]]+	one or more none-ASCII characters (For example, Unicode charac
[\n\t]+	one or more {newline character, tab, space}.

Demo cu Emacs

(Mathematical) Definition

Definition

The set of *regular expressions* over the alphabet Σ is is recursively defined as follows:

(Mathematical) Definition

Definition

The set of *regular expressions* over the alphabet Σ is is recursively defined as follows:

(Mathematical) Definition

Definition

The set of *regular expressions* over the alphabet Σ is is recursively defined as follows:

- ε, *empty* are regular expressions

(Mathematical) Definition

Definition

The set of *regular expressions* over the alphabet Σ is is recursively defined as follows:

- ε, *empty* are regular expressions
- any character in Σ is a regular expression;

(Mathematical) Definition

Definition

The set of *regular expressions* over the alphabet Σ is is recursively defined as follows:

- ε, *empty* are regular expressions
- any character in Σ is a regular expression;
- if e_1, e_2 are regular expressions, then $e_1 e_2$ is a regular expression;

(Mathematical) Definition

Definition

The set of *regular expressions* over the alphabet Σ is is recursively defined as follows:

- ε, *empty* are regular expressions
- any character in Σ is a regular expression;
- if e_1, e_2 are regular expressions, then $e_1 e_2$ is a regular expression;
- if e_1, e_2 are regular expressions, then $e_1 + e_2$ is a regular expression;

(Mathematical) Definition

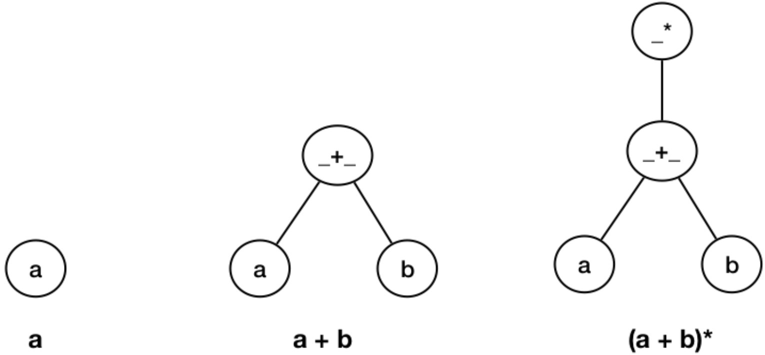
Definition

The set of *regular expressions* over the alphabet Σ is is recursively defined as follows:

- ε, *empty* are regular expressions
- any character in Σ is a regular expression;
- if e_1, e_2 are regular expressions, then $e_1 e_2$ is a regular expression;
- if e_1, e_2 are regular expressions, then $e_1 + e_2$ is a regular expression;
- if e is a regular expression, then e^* is a regular expressions.

Often we use parentheses to show how the above rules were applied; e.g., $(a + b)^*$

Abstract Syntactic Tree (AST)



Relationship with the package <regex> in C++, Emacs

<regex>	regular expression in math notation
[abc]	$a + b + c$
[0-9]	$0 + 1 + \cdots + 9$
[0-9]*	$(0 + 1 + \cdots + 9)^*$
[0-9]+	$(0 + 1 + \cdots + 9)(0 + 1 + \cdots + 9)^*$

The language defined by a regular expression 1/2

Definition

The set of strings (language) $L(e)$ described by a regular expression is recursively defined as follows:

The language defined by a regular expression 1/2

Definition

The set of strings (language) $L(e)$ described by a regular expression is recursively defined as follows:

- $L(\varepsilon) =$

The language defined by a regular expression 1/2

Definition

The set of strings (language) $L(e)$ described by a regular expression is recursively defined as follows:

- $L(\varepsilon) = \{\varepsilon\}$

The language defined by a regular expression 1/2

Definition

The set of strings (language) $L(e)$ described by a regular expression is recursively defined as follows:

- $L(\varepsilon) = \{\varepsilon\}$ (ε is the empty string (of size zero)),

The language defined by a regular expression 1/2

Definition

The set of strings (language) $L(e)$ described by a regular expression is recursively defined as follows:

- $L(\varepsilon) = \{\varepsilon\}$ (ε is the empty string (of size zero)),
- $L(\texttt{empty}) =$

The language defined by a regular expression 1/2

Definition

The set of strings (language) $L(e)$ described by a regular expression is recursively defined as follows:

- $L(\varepsilon) = \{\varepsilon\}$ (ε is the empty string (of size zero)),
- $L(\mathtt{empty}) = \emptyset$

The language defined by a regular expression 1/2

Definition

The set of strings (language) $L(e)$ described by a regular expression is recursively defined as follows:

- $L(\varepsilon) = \{\varepsilon\}$ (ε is the empty string (of size zero)),
- $L(\texttt{empty}) = \emptyset$
- if e is un character

The language defined by a regular expression 1/2

Definition

The set of strings (language) $L(e)$ described by a regular expression is recursively defined as follows:

- $L(\varepsilon) = \{\varepsilon\}$ (ε is the empty string (of size zero)),
- $L(\texttt{empty}) = \emptyset$
- if e is un character then $L(e) = \{e\}$;

The language defined by a regular expression 1/2

Definition

The set of strings (language) $L(e)$ described by a regular expression is recursively defined as follows:

- $L(\varepsilon) = \{\varepsilon\}$ (ε is the empty string (of size zero)),
- $L(\texttt{empty}) = \emptyset$
- if e is un character then $L(e) = \{e\}$;
- if $e = e_1 e_2$

The language defined by a regular expression 1/2

Definition

The set of strings (language) $L(e)$ described by a regular expression is recursively defined as follows:

- $L(\varepsilon) = \{\varepsilon\}$ (ε is the empty string (of size zero)),
- $L(\texttt{empty}) = \emptyset$
- if e is un character then $L(e) = \{e\}$;
- if $e = e_1 e_2$ then $L(e) = L(e_1)L(e_2) = \{w_1 w_2 \mid w_1 \in L(e_1), w_2 \in L(e_2)\}$;

The language defined by a regular expression 1/2

Definition

The set of strings (language) $L(e)$ described by a regular expression is recursively defined as follows:

- $L(\varepsilon) = \{\varepsilon\}$ (ε is the empty string (of size zero)),
- $L(\texttt{empty}) = \emptyset$
- if e is un character then $L(e) = \{e\}$;
- if $e = e_1 e_2$ then $L(e) = L(e_1)L(e_2) = \{w_1 w_2 \mid w_1 \in L(e_1), w_2 \in L(e_2)\}$;
- if $e = e_1 + e_2$

The language defined by a regular expression 1/2

Definition

The set of strings (language) $L(e)$ described by a regular expression is recursively defined as follows:

- $L(\varepsilon) = \{\varepsilon\}$ (ε is the empty string (of size zero)),
- $L(\texttt{empty}) = \emptyset$
- if e is un character then $L(e) = \{e\}$;
- if $e = e_1 e_2$ then $L(e) = L(e_1)L(e_2) = \{w_1 w_2 \mid w_1 \in L(e_1), w_2 \in L(e_2)\}$;
- if $e = e_1 + e_2$ then $L(e) = L(e_1) \cup L(e_2)$;

The language defined by a regular expression 1/2

Definition

The set of strings (language) $L(e)$ described by a regular expression is recursively defined as follows:

- $L(\varepsilon) = \{\varepsilon\}$ (ε is the empty string (of size zero)),
- $L(\mathtt{empty}) = \emptyset$
- if e is un character then $L(e) = \{e\}$;
- if $e = e_1 e_2$ then $L(e) = L(e_1)L(e_2) = \{w_1 w_2 \mid w_1 \in L(e_1), w_2 \in L(e_2)\}$;
- if $e = e_1 + e_2$ then $L(e) = L(e_1) \cup L(e_2)$;
- if $e = e_1^*$

The language defined by a regular expression 1/2

Definition

The set of strings (language) $L(e)$ described by a regular expression is recursively defined as follows:

- $L(\varepsilon) = \{\varepsilon\}$ (ε is the empty string (of size zero)),
- $L(\texttt{empty}) = \emptyset$
- if e is un character then $L(e) = \{e\}$;
- if $e = e_1 e_2$ then $L(e) = L(e_1)L(e_2) = \{w_1 w_2 \mid w_1 \in L(e_1), w_2 \in L(e_2)\}$;
- if $e = e_1 + e_2$ then $L(e) = L(e_1) \cup L(e_2)$;
- if $e = e_1^*$ then $L(e) = \cup_k L(e_1^k)$, where $L(e_1^0) = \{\varepsilon\}, L(e_1^{k+1}) = L(e_1)L(e_1^k)$;

The language defined by a regular expression 1/2

Definition

The set of strings (language) $L(e)$ described by a regular expression is recursively defined as follows:

- $L(\varepsilon) = \{\varepsilon\}$ (ε is the empty string (of size zero)),
- $L(\texttt{empty}) = \emptyset$
- if e is un character then $L(e) = \{e\}$;
- if $e = e_1 e_2$ then $L(e) = L(e_1)L(e_2) = \{w_1 w_2 \mid w_1 \in L(e_1), w_2 \in L(e_2)\}$;
- if $e = e_1 + e_2$ then $L(e) = L(e_1) \cup L(e_2)$;
- if $e = e_1^*$ then $L(e) = \cup_k L(e_1^k)$, where $L(e_1^0) = \{\varepsilon\}, L(e_1^{k+1}) = L(e_1)L(e_1^k)$;
- if $e = (e_1)$

The language defined by a regular expression 1/2

Definition

The set of strings (language) $L(e)$ described by a regular expression is recursively defined as follows:

- $L(\varepsilon) = \{\varepsilon\}$ (ε is the empty string (of size zero)),
- $L(\mathtt{empty}) = \emptyset$
- if e is un character then $L(e) = \{e\}$;
- if $e = e_1 e_2$ then $L(e) = L(e_1)L(e_2) = \{w_1 w_2 \mid w_1 \in L(e_1), w_2 \in L(e_2)\}$;
- if $e = e_1 + e_2$ then $L(e) = L(e_1) \cup L(e_2)$;
- if $e = e_1^*$ then $L(e) = \cup_k L(e_1^k)$, where $L(e_1^0) = \{\varepsilon\}, L(e_1^{k+1}) = L(e_1)L(e_1^k)$;
- if $e = (e_1)$ then $L(e) = L(e_1)$.

Remark. The operator $\_^*$ is called Kleene star or Kleene closure.

The language defined by a regular expression 2/2

Example: Consider $A = \{a, b, c\}$. We have
$L(a(b + a)c) =$

The language defined by a regular expression 2/2

Example: Consider $A = \{a, b, c\}$. We have
$L(a(b + a)c) = \{abc, aac\}$ and
$L((ab)^*) =$

The language defined by a regular expression 2/2

Example: Consider $A = \{a, b, c\}$. We have
$L(a(b + a)c) = \{abc, aac\}$ and
$L((ab)^*) = \{\varepsilon, ab, abab, ababab, \ldots\} = \{(ab)^k \mid k \geq 0\}$.
Justification:

The language defined by a regular expression 2/2

Example: Consider $A = \{a, b, c\}$. We have
$L(a(b + a)c) = \{abc, aac\}$ and
$L((ab)^*) = \{\varepsilon, ab, abab, ababab, \ldots\} = \{(ab)^k \mid k \geq 0\}$.
Justification:
$L(a) = \{a\}$

The language defined by a regular expression 2/2

Example: Consider $A = \{a, b, c\}$. We have
$L(a(b + a)c) = \{abc, aac\}$ and
$L((ab)^*) = \{\varepsilon, ab, abab, ababab, \ldots\} = \{(ab)^k \mid k \geq 0\}$.
Justification:
$L(a) = \{a\}$
$L(b) = \{b\}$

The language defined by a regular expression 2/2

Example: Consider $A = \{a, b, c\}$. We have
$L(a(b + a)c) = \{abc, aac\}$ and
$L((ab)^*) = \{\varepsilon, ab, abab, ababab, \ldots\} = \{(ab)^k \mid k \geq 0\}$.
Justification:
$L(a) = \{a\}$
$L(b) = \{b\}$
$L(b + a) = L(b) \cup L(a) = \{b, a\}$

The language defined by a regular expression 2/2

Example: Consider $A = \{a, b, c\}$. We have
$L(a(b + a)c) = \{abc, aac\}$ and
$L((ab)^*) = \{\varepsilon, ab, abab, ababab, \ldots\} = \{(ab)^k \mid k \geq 0\}$.
Justification:
$L(a) = \{a\}$
$L(b) = \{b\}$
$L(b + a) = L(b) \cup L(a) = \{b, a\}$
$L(a(b + a)) = L(a)L(b + a) = \{ab, aa\}$

The language defined by a regular expression 2/2

Example: Consider $A = \{a, b, c\}$. We have
$L(a(b+a)c) = \{abc, aac\}$ and
$L((ab)^*) = \{\varepsilon, ab, abab, ababab, \ldots\} = \{(ab)^k \mid k \geq 0\}$.
Justification:
$L(a) = \{a\}$
$L(b) = \{b\}$
$L(b+a) = L(b) \cup L(a) = \{b, a\}$
$L(a(b+a)) = L(a)L(b+a) = \{ab, aa\}$
$L(c) = \{c\}$

The language defined by a regular expression 2/2

Example: Consider $A = \{a, b, c\}$. We have
$L(a(b + a)c) = \{abc, aac\}$ and
$L((ab)^*) = \{\varepsilon, ab, abab, ababab, \ldots\} = \{(ab)^k \mid k \geq 0\}$.
Justification:
$L(a) = \{a\}$
$L(b) = \{b\}$
$L(b + a) = L(b) \cup L(a) = \{b, a\}$
$L(a(b + a)) = L(a)L(b + a) = \{ab, aa\}$
$L(c) = \{c\}$
$L(a(b + a)c) = L(a(b + a))L(c) = \{abc, aac\}$

The language defined by a regular expression 2/2

Example: Consider $A = \{a, b, c\}$. We have
$L(a(b+a)c) = \{abc, aac\}$ and
$L((ab)^*) = \{\varepsilon, ab, abab, ababab, \ldots\} = \{(ab)^k \mid k \geq 0\}$.
Justification:
$L(a) = \{a\}$
$L(b) = \{b\}$
$L(b+a) = L(b) \cup L(a) = \{b, a\}$
$L(a(b+a)) = L(a)L(b+a) = \{ab, aa\}$
$L(c) = \{c\}$
$L(a(b+a)c) = L(a(b+a))L(c) = \{abc, aac\}$
$L((ab)^0) = \{\varepsilon\}$

The language defined by a regular expression 2/2

Example: Consider $A = \{a, b, c\}$. We have
$L(a(b + a)c) = \{abc, aac\}$ and
$L((ab)^*) = \{\varepsilon, ab, abab, ababab, \ldots\} = \{(ab)^k \mid k \geq 0\}$.
Justification:
$L(a) = \{a\}$
$L(b) = \{b\}$
$L(b + a) = L(b) \cup L(a) = \{b, a\}$
$L(a(b + a)) = L(a)L(b + a) = \{ab, aa\}$
$L(c) = \{c\}$
$L(a(b + a)c) = L(a(b + a))L(c) = \{abc, aac\}$
$L((ab)^0) = \{\varepsilon\}$
$L((ab)^1) = L(ab) = \{ab\}$

The language defined by a regular expression 2/2

Example: Consider $A = \{a, b, c\}$. We have
$L(a(b+a)c) = \{abc, aac\}$ and
$L((ab)^*) = \{\varepsilon, ab, abab, ababab, \ldots\} = \{(ab)^k \mid k \geq 0\}$.
Justification:
$L(a) = \{a\}$
$L(b) = \{b\}$
$L(b+a) = L(b) \cup L(a) = \{b, a\}$
$L(a(b+a)) = L(a)L(b+a) = \{ab, aa\}$
$L(c) = \{c\}$
$L(a(b+a)c) = L(a(b+a))L(c) = \{abc, aac\}$
$L((ab)^0) = \{\varepsilon\}$
$L((ab)^1) = L(ab) = \{ab\}$
$L((ab)^2) = L(ab)L(ab) = \{abab\}$

The language defined by a regular expression 2/2

Example: Consider $A = \{a, b, c\}$. We have
$L(a(b + a)c) = \{abc, aac\}$ and
$L((ab)^*) = \{\varepsilon, ab, abab, ababab, \ldots\} = \{(ab)^k \mid k \geq 0\}$.
Justification:
$L(a) = \{a\}$
$L(b) = \{b\}$
$L(b + a) = L(b) \cup L(a) = \{b, a\}$
$L(a(b + a)) = L(a)L(b + a) = \{ab, aa\}$
$L(c) = \{c\}$
$L(a(b + a)c) = L(a(b + a))L(c) = \{abc, aac\}$
$L((ab)^0) = \{\varepsilon\}$
$L((ab)^1) = L(ab) = \{ab\}$
$L((ab)^2) = L(ab)L(ab) = \{abab\}$
$L((ab)^3) = L(ab)L((ab)^2) = \{ababab\}$

The language defined by a regular expression 2/2

Example: Consider $A = \{a, b, c\}$. We have
$L(a(b + a)c) = \{abc, aac\}$ and
$L((ab)^*) = \{\varepsilon, ab, abab, ababab, \ldots\} = \{(ab)^k \mid k \geq 0\}$.
Justification:
$L(a) = \{a\}$
$L(b) = \{b\}$
$L(b + a) = L(b) \cup L(a) = \{b, a\}$
$L(a(b + a)) = L(a)L(b + a) = \{ab, aa\}$
$L(c) = \{c\}$
$L(a(b + a)c) = L(a(b + a))L(c) = \{abc, aac\}$
$L((ab)^0) = \{\varepsilon\}$
$L((ab)^1) = L(ab) = \{ab\}$
$L((ab)^2) = L(ab)L(ab) = \{abab\}$
$L((ab)^3) = L(ab)L((ab)^2) = \{ababab\}$
...
$L((ab)^*) = \bigcup_{k \geq 0} L((ab)^k) = \{\varepsilon, ab, abab, ababab, \ldots\}$

The end

The next lecture:

- the AST of regular expression in ALK
- parsing algorithm
- the automaton associated to a regular expression
- searching using the automaton