Pretraživanje stringova

Strukture podataka i algoritmi 2

Pretraživanje stringova

- **Problem: da li se string *patern* sadrži u stringu *tekst* kao podstring?**
  - Algoritmi za pretraživanje stringova (*string matching*)
- **Primene**
  - Pretraživanje tekstualnog sadržaja, CTRL+F u editorima teksta i mnogim drugim softverima
  - Detekcija plagijata
  - Bioinformatika
    - DNK sekvenca je string formiran od slova A, C, G i T (DNK alfabet)
    - Da li se jedna DNK sekvenca pojavljuje u nekoj drugoj DNK sekvenci?

Pretraživanje stringova

- Problem: da li se string patern sadrži u stringu tekst kao podstring?
  - Ako se patern pojavljuje u tekstu onda je potrebno vratiti i poziciju prve pojave. Inače vraćamo -1.
  - Rešenje prethodnog problema se trivijalno uopštava na sve pojave paterna u tekstu.

```
String pattern = "ana";
String text = "banana voli milovana";

// public int indexOf(String str, int fromIndex)
// ----------------------------------------
// returns the index within this string of the first occurrence
// of the specified substring, starting at the specified index
int pos = text.indexOf(pattern, 0);
while (pos != -1) {
	System.out.println(pos);
	pos = text.indexOf(pattern, pos + 1);
}
```

Pretraživanje grubom silom

- Ideja: za svaki podstring stringa tekst proveriti da li je jednak sa patern
  1. Poravnamo *patern* sa početkom stringa *tekst*
  2. Proveravamo da li se poravnati znaci poklapaju
  3. Ako se svi znaci poklapaju našli smo prvu pojavu *patern* u tekstu
  4. Kod prvog nepoklapanja poravnatih znakova pomerimo *patern* za jedno mesto u desno i vratimo se na korak 2 (imamo novo poravnanje)

Pretraživanje stringova grubom silom

- patern = “ana”
- tekst = “dani banalni”

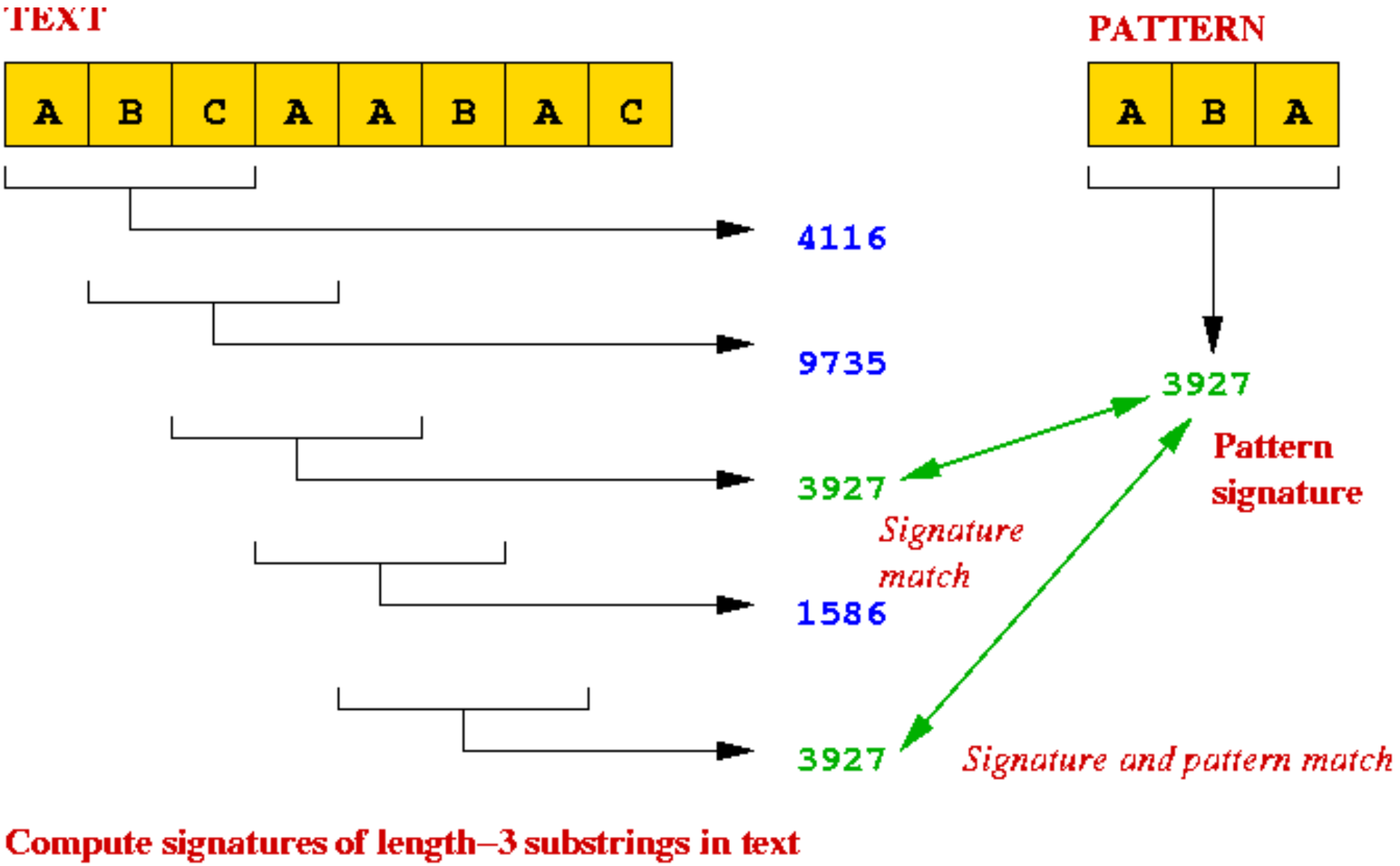
```
dani banalni
ana                  ana poravnato sa dan
 ana                 ana poravnato sa ani
  ana                ana poravnato sa ni
   ana               ana poravnato sa i b
    ana              ana poravnato sa  ba
     ana             ana poravnato sa ban
      ana            ana poravnato sa ana
```

```
public static int search(String pat, String txt) {
	if (pat.length() > txt.length())
		throw new IllegalArgumentException(
			"Patern veci od teksta");

	for (int i = 0; i <= txt.length() - pat.length(); i++) {
		boolean match = true;
		int j = 0;
		while (match && j < pat.length()) {
			if (pat.charAt(j) != txt.charAt(i + j)) {
				match = false;
			} else {
				j++;
			}
		}

		if (match)
			return i;
	}

	return -1;
}
```

Rabin-Karpov algoritam

- Ideja:
  - **Za svaki string možemo izračunati heš kod**
    - Heš kod kao *fingerprint* (*signature*) koji ćemo iskoristiti za preliminarno poređenje stringova
  - **HP** = heš kod stringa *patern*
  - Za proizvoljno poravnanje izračunamo heš kod poravnanjem obuhvaćenog podstringa *tekst* (**HT**)
  - Ako se **HP** i **HT** razlikuju tada sigurno znamo da nemamo poklapanje svih poravnatih karaktera
    - **Posledica konzistentnosti heš funkcije**
  - Inače poredimo znak po znak kao kod pretraživanja stringova grubom silom

Rabin-Karpov algoritam



Heš kod za klasu String

```
public final class String
{
   private final char[] s;
   ...

   public int hashCode()
   {
      int hash = 0;
      for (int i = 0; i < length(); i++)
         hash = s[i] + (31 * hash);
      return hash;
   }
}
```

- L = length() - 1
- hash = $s[0] \cdot 31^{L} + s[1] \cdot 31^{L-1} + s[2] \cdot 31^{L-2} + \ldots + s[L-1] \cdot 31 + s[L]$

Trik kod RK algoritma

- **Ako bi smo za svaki podstring stringa *tekst* računali heš kod tada RK ne bi bio efikasniji od algoritma grube sile.**
  - Složenost O(*PT*) gde je *P* dužina paterna, a *T* dužina teksta

- Ideja je da heš kod podstringa *tekst* ažuiramo u O(1) pošto napravimo novo poravnanje
  - Podstring stringa tekst se novim poravnanjem menja brisanjem prvog slova i dodavanjem novog slova na kraj
  - Ograničićemo vrednost heš funkcije u opsegu [0 .. LP], gde je LP neki veliki prost broj, kako bi smo mogli izvesti ažuriranje
    - **31^L može biti veće od Integer.MAX_VALUE za dugačak *patern***

Ažuriranje heš vrednosti

- Neka je string *s* sačinjen od karaktera $C_0, C_1 \ldots, C_L$

$$hash(s) = (C_0 \cdot 31^L + C_1 \cdot 31^{L-1} + \ldots + C_{L-1} \cdot 31 + C_L) \% LP$$

- Neka se *s'* dobija od *s* brisanjem C_0 i dodavanjem novog karaktera B na kraj

$$hash(s') = (C_1 \cdot 31^L + C_2 \cdot 31^{L-1} + \ldots + C_{L-1} \cdot 31^2 + C_L \cdot 31 + B) \% LP$$

- hash(s') možemo dobiti iz hash(s) na sledeći način:
  - X = odstranimo karakter C_0 iz hash(s)

$$X = (hash(s) - (C_0 \cdot (31^L \% LP)) \% LP) \% LP$$

hash(s) **može biti manje od** $Y = (C_0 \cdot (31^L \% LP)) \% LP$

$$X = (LP + hash(s) - Y) \% LP$$

  - Pomnožimo X sa 31 i dodamo B

$$hash(s') = ((31 \cdot X) \% LP + B) \% LP$$

$31^L \% LP$ je konstanta koju ćemo izračunati na početku programa kako je ne bi računali svaki put iznova

```java
public class RKSearch {
	private static final int LP = 15485863;
	private String pat, txt;
	private int patH;          // hash od pattern
	private int txtH;          // hash od podstringa text
	private int fcFactor;      // 31^L % LP

	public RKSearch(String pat, String txt) {
		if (pat.length() > txt.length())
			throw new IllegalArgumentException(
					"Patern veci od teksta");

		this.pat = pat;
		this.txt = txt;

		patH = computeHash(pat, pat.length());
		txtH = computeHash(txt, pat.length());

		fcFactor = 1;
		for (int i = 0; i < pat.length() - 1; i++) {
			fcFactor = (fcFactor * 31) % LP;
		}
	}
	...
}
```

Heš funkcija i njeno ažuriranje

```
/* Racuna hash za prvih len karaktera str */
private int computeHash(String str, int len) {
	int hash = 0;
	for (int i = 0; i < len; i++) {
		hash = ((31 * hash) % LP + str.charAt(i)) % LP;
	}

	return hash;
}

/*
 * Azurira vrednost hesh funkciju za podstring txt
 * tako sto brise karakter na poziciji pos i dodaje
 * prvi sledeci karakter
 * (koji je na poziciji pos + pat.length)
 */
private void updateTextHash(int pos) {
	txtH = (LP + txtH - (txt.charAt(pos) * fcFactor) % LP) % LP;
	txtH = ((31 * txtH) % LP + txt.charAt(pos + pat.length())) % LP;
}
```

RK algoritam

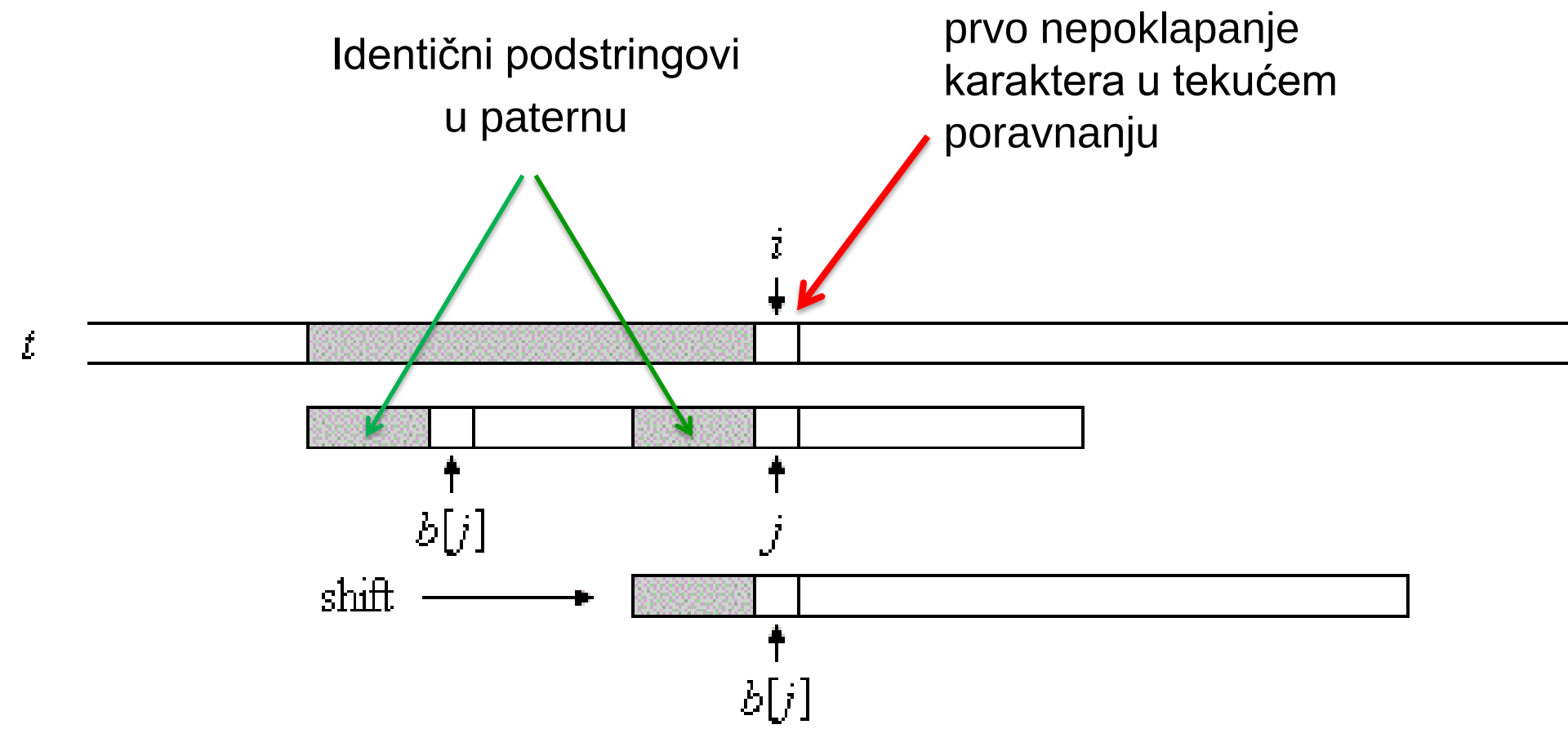
```
public int search() {
	int lastAlignment = txt.length() - pat.length();
	for (int i = 0; i <= lastAlignment; i++) {
		if (patH == txtH) {
			boolean match = true;
			int j = 0;
			while (match && j < pat.length()) {
				if (pat.charAt(j) != txt.charAt(i + j)) {
					match = false;
				} else {
					j++;
				}
			}

			if (match)
				return i;
		}
		if (i < lastAlignment) updateTextHash(i);
	}

	return -1;
}
```

Knut-Moris-Prat (KMP) algoritam

- 1970. – Knut osmislio KMP algoritam baziran na primeni deterministčkih konačnih automata
  - Potom je algoritam predstavio Pratu koji ga je unapredio
- 1969. – Moris je radeći na tekst editoru za CDC 6400 računar, pre i nezavisno od Knuta, osmislio algoritam sličan Knutovom
  - Prat je Knutov originalni algoritam i svoju modifikaciju prezentovao Morisu kad ovaj shvata...
- **1977. – Knut, Moris i Prat publikuju zajednički rad gde predstavljaju KMP algoritam u Pratovoj verziji**

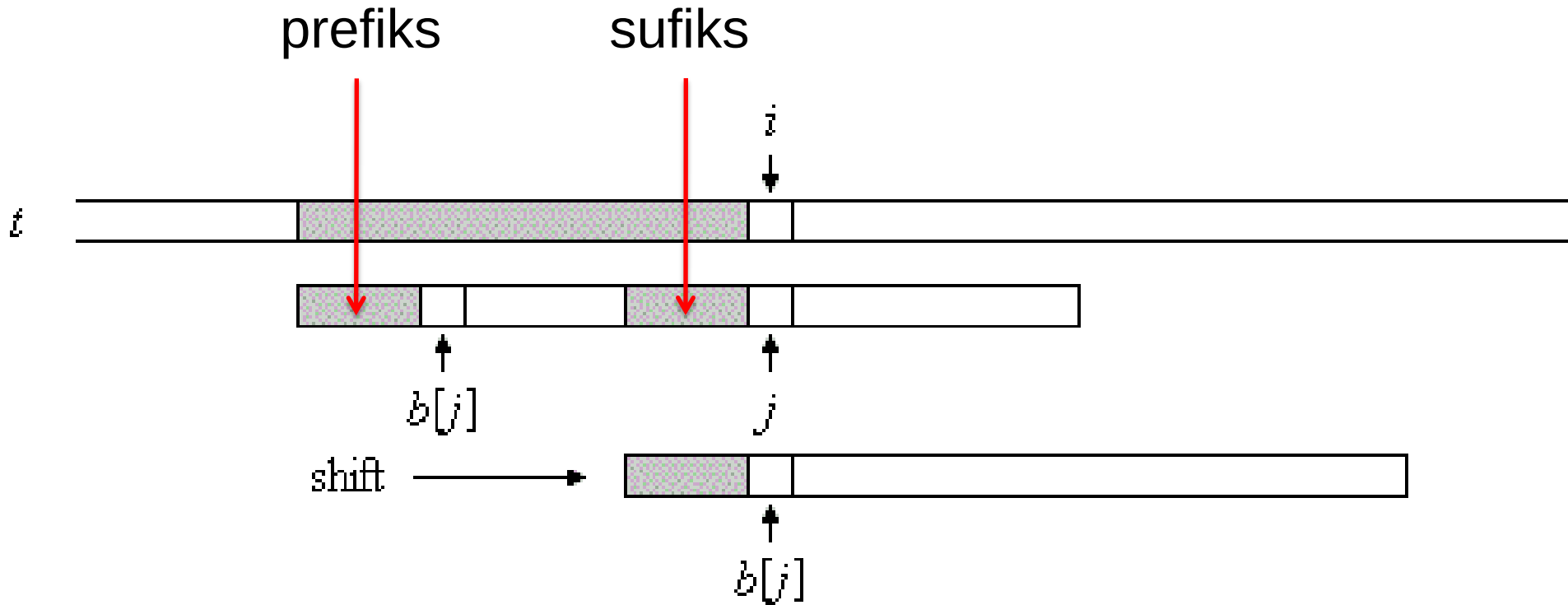
KMP algoritam, Pratova verzija

- Ideja: pomeriti *patern* za više od jednog mesta u desno ukoliko je to moguće

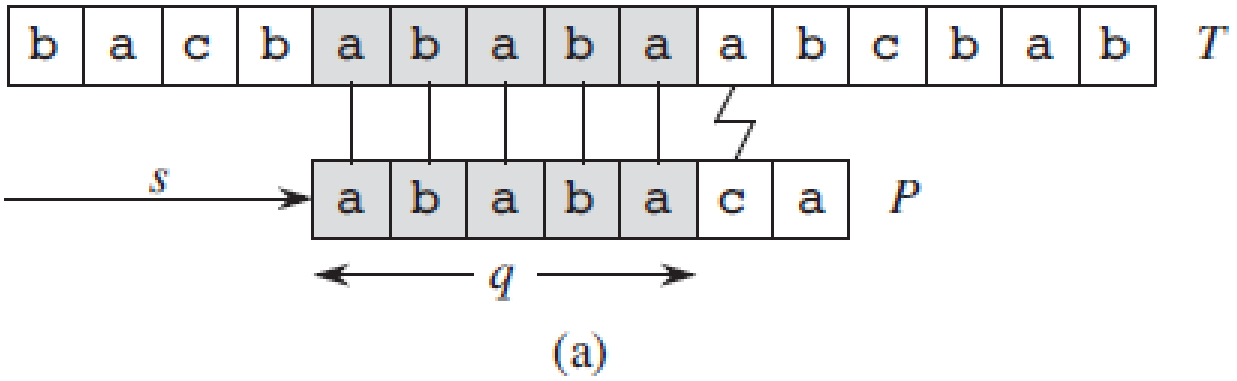


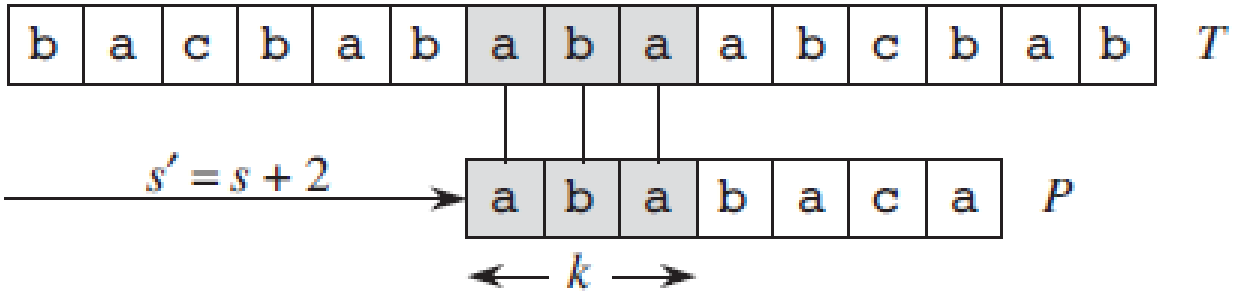
Identični podstringovi

- Neka je P deo paterna koji se poklopio sa tekstom
- Pomeranje paterna za više mesta u desno je moguće ako je neki **pravi prefiks** P identičan sa **pravim sufiksom** P iste dužine.
  - Stoga za svaki karakter u paternu moramo izračunati dužinu **maksimalnog pravog prefiksa** koji se poklapa sa pravim sufiksom iste dužine.



Maksimalni pravi prefiks može da se preklapa sa pravim sufiksom





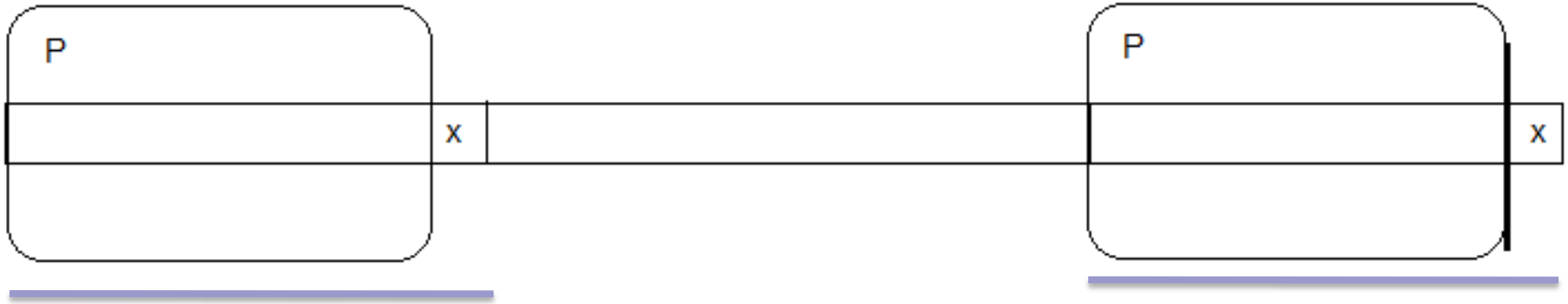
U stringu “ababa” pravi prefiks “aba” se preklapa sa pravim sufiksom “aba”

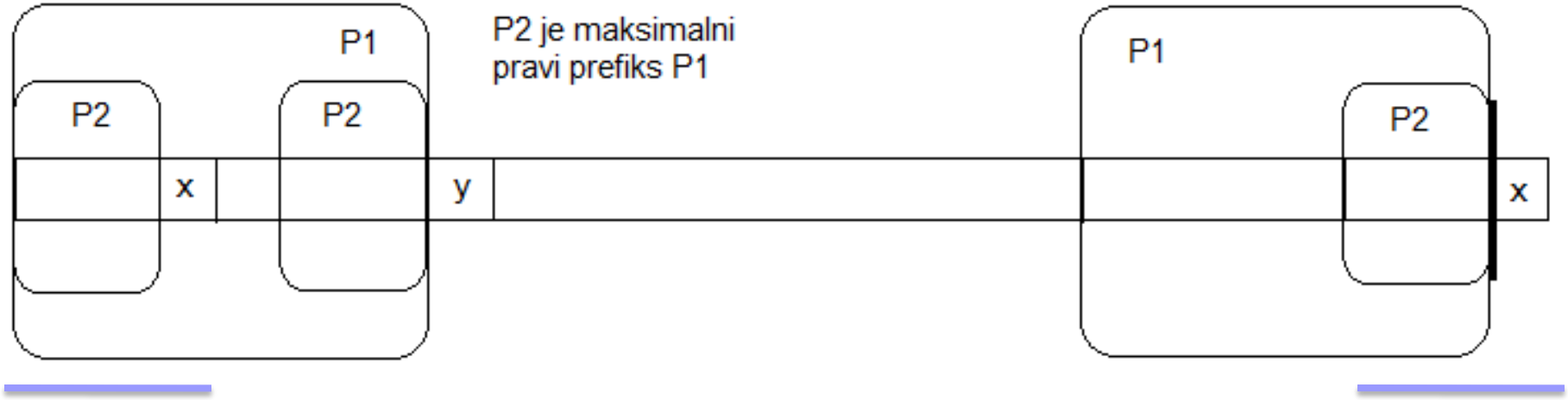
Tabela prefiksa

- Tabela prefiksa je niz ***prefix*** koji se računa na osnovu paterna i ima sledeće osobine
  - Dužina niza ***prefix*** je dužina paterna plus jedan
  - **prefix[i]** → dužina maksimalnog pravog prefiksa koji je identičan maksimalnom pravom sufiksu za ***patern*****[0 .. i – 1]**
  - prefix[0] = -1

indeks	0	1	2	3	4	5	6
pattern	A	N	A	N	A	A	
prefix	-1	0	0	1	2	3	1

Računanje tabele prefiksa





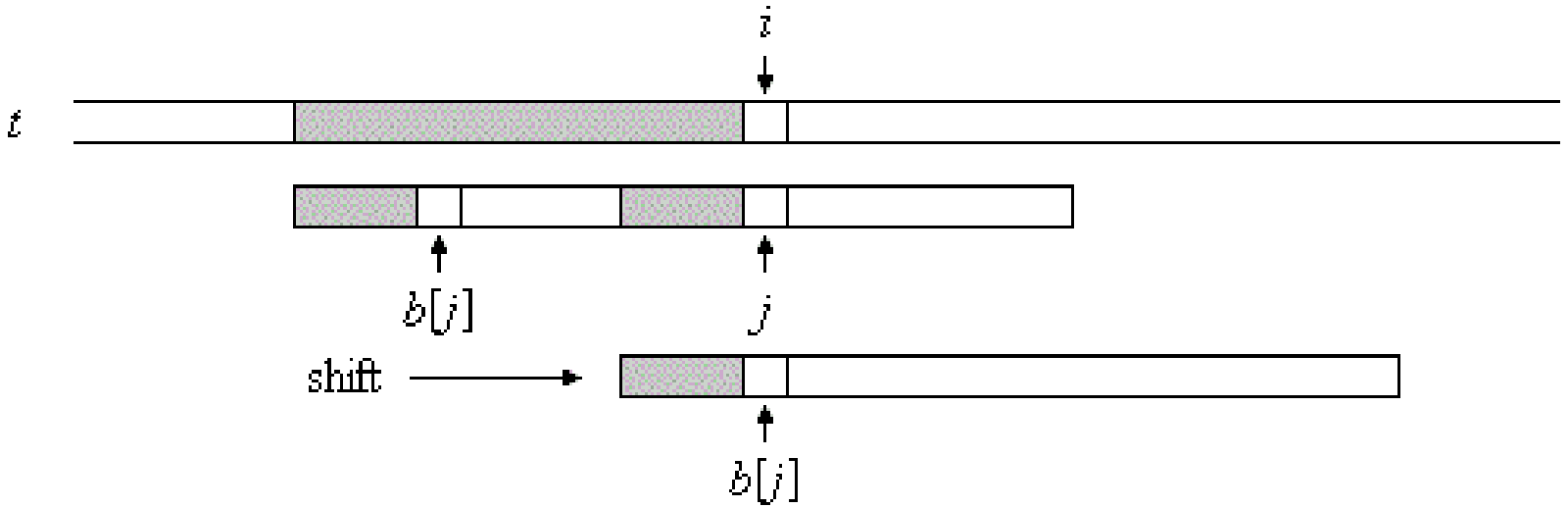
Propagacija po prefiksima ka početku

- **prefix[i] se odnosi na string pattern[0 .. i – 1]**
- prefix[i] = prefix[i – 1] + 1
  ako pattern[i – 1] = pattern[prefix[i – 1]]
- prefix[i] = prefix[prefix[i – 1]] + 1
  ako pattern[i – 1] = pattern[prefix[prefix[i – 1]]
- prefix[i] = prefix[prefix[prefix[i – 1]]] + 1
  ako pattern[i – 1] = pattern[prefix[prefix[prefix[i – 1]]]]
- ...
- prefix[i] = prefix[k] + 1 ako pattern[i – 1] = pattern[k]
  gde se k menja na način k = prefix[k]
  za početno k = prefix[i – 1]

```
public class KMPSearch {
	private String pat;
	private int[] prefix;

	public KMPSearch(String pat) {
		this.pat = pat;
		buildPrefixTable();
	}

	private void buildPrefixTable() {
		prefix = new int[pat.length() + 1];
		prefix[0] = -1;

		for (int i = 1; i <= pat.length(); i++) {
			int k = prefix[i - 1];
			char c = pat.charAt(i - 1);

			while (k >= 0 && c != pat.charAt(k)) {
				k = prefix[k];
			}

			prefix[i] = k + 1;
		}

	}
	 ...
}
```

KMP algoritam

- Dva indeksa
  - *i* – pokazuje na trenutni karakter u *text*
    (i: 0 → text.length() – 1)
  - *j* – pokazuje na trenutni karakter u *pattern*
- Kod nepoklapanja karaktera **text[i]** i **pattern[j]** razlikujemo dva slučaja
  - **j = 0** → *i* se uvećava za 1
  - **j > 0** → ***i* se ne menja**, **j = prefix[j]**



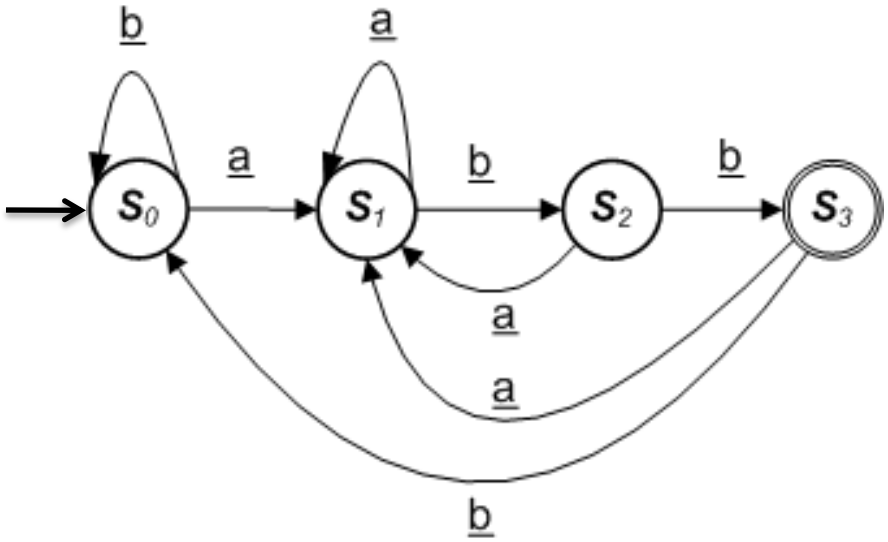
```
public class KMPSearch {
	...
	public int searchIn(String txt) {
		int i = 0, j = 0;

		while (i + pat.length() - j <= txt.length()) {
			boolean match = true;
			while (match && j < pat.length()) {
				if (pat.charAt(j) != txt.charAt(i)) {
					match = false;
				} else {
					j++; i++;
				}
			}

			if (match) {
				return i - pat.length();
			}
			else {
				if (j == 0) i++;
				else j = prefix[j];
			}
		}

		return -1;
	}
}
```

KMP algoritam – DFA verzija

- Ideja:
  - propustiti tekst kroz **deterministički konačni automat (DFA)** koji je konstruisan na osnovu paterna
  - DFA – apstraktna mašina koja na ulazu prima jedan simbol (karakter) i sastoji se od stanja i tranzicione funkcije
    - DFA je uvek u nekom stanju
      - Početno stanje (stanje pre prvog ulaza)
      - Finalno stanje (stanje u kome je mašina prepoznala patern)
      - Ostala stanja (stanja u kome je mašina prepoznala delove paterna)
    - Tranziciona funkcija definiše promenu stanja automata – određuje sledeće stanje na osnovu trenutnog stanja i simbola na ulazu

DFA ljudima predstavljamo grafički

- Stanja crtamo kružićima (finalno stanje dva kružića)
- Tranzicionu funkciju labeliranim strelicama koje označavaju promenu jednog stanja u neko drugo na osnovu simbola sa ulaza



Pravila automata sa slike:

Ako je DFA u stanju S_0 i na ulazu dobije a tada prelazi u stanje S_1

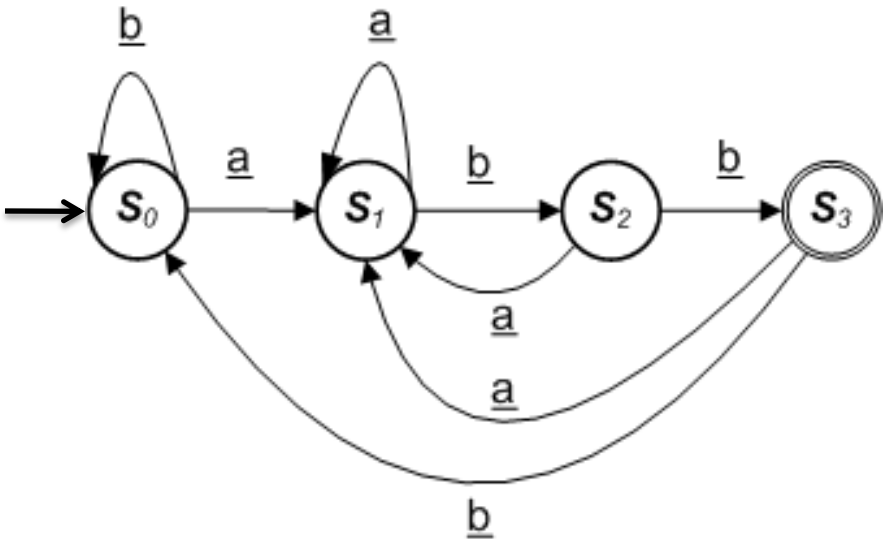
Ako je DFA u stanju S_0 i na ulazu dobije b tada ostaje u stanju S_0

Ako je DFA u stanju S_1 i na ulazu dobije b tada prelazi u stanje S_2

...

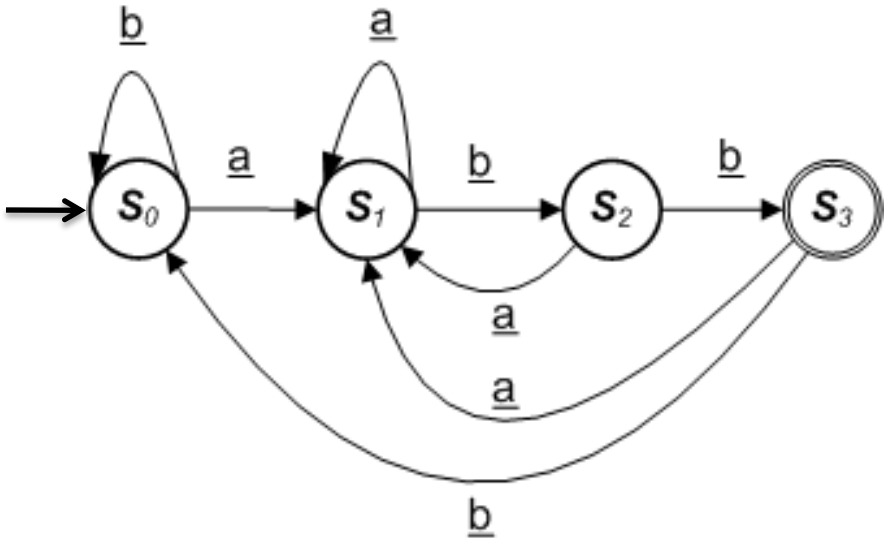
Ako je DFA u stanju u S_3 i na ulazu dobije b tada prelazi u stanje u S_0

DFA računaru predstavljamo matricom koja opisuje tranzicionu funkciju



Ulaz \ Stanje	0	1	2	3
a	1	1	1	1
b	0	2	3	0

dfa[X][Y] = Z → automat u stanju Y prelazi u stanje Z na ulaz X



Simulacija rada automata za ulaz “bababb”

S = 0, input = b → S = 0

S = 0, input = a → S = 1

S = 1, input = b → S = 2

S = 2, input = a → S = 1

S = 1, input = b → S = 2

S = 2, input = b → S = 3

Koje reči prepoznaje automat sa gornje slike?

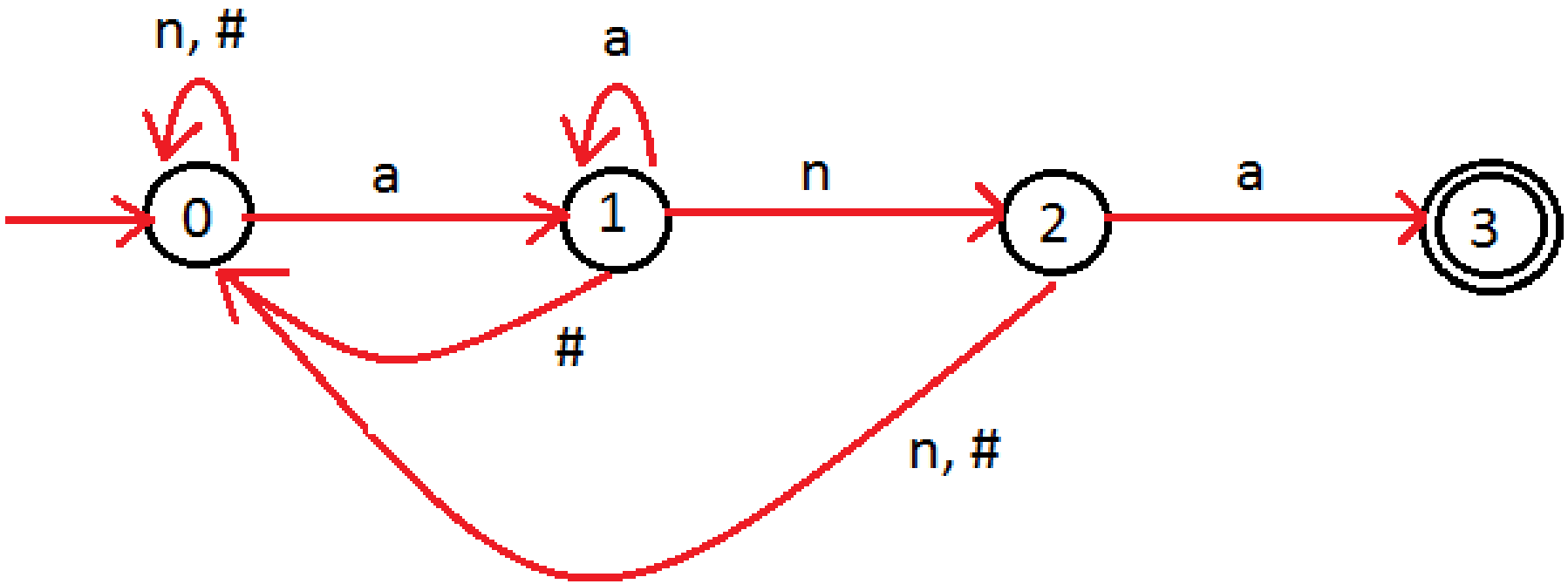
Simulacija DFA

Pseudo-kod koji opisuje rad DFA:

```
int state = startState;
while (state != finalState) {
    c = read input symbol;
    state = dfa[c][state];
}
```

gde je dfa matrica koja opisuje tranzicionu funkciju

Konstruisati konačan automat koji prepoznaje reči koje sadrže patern “ana”

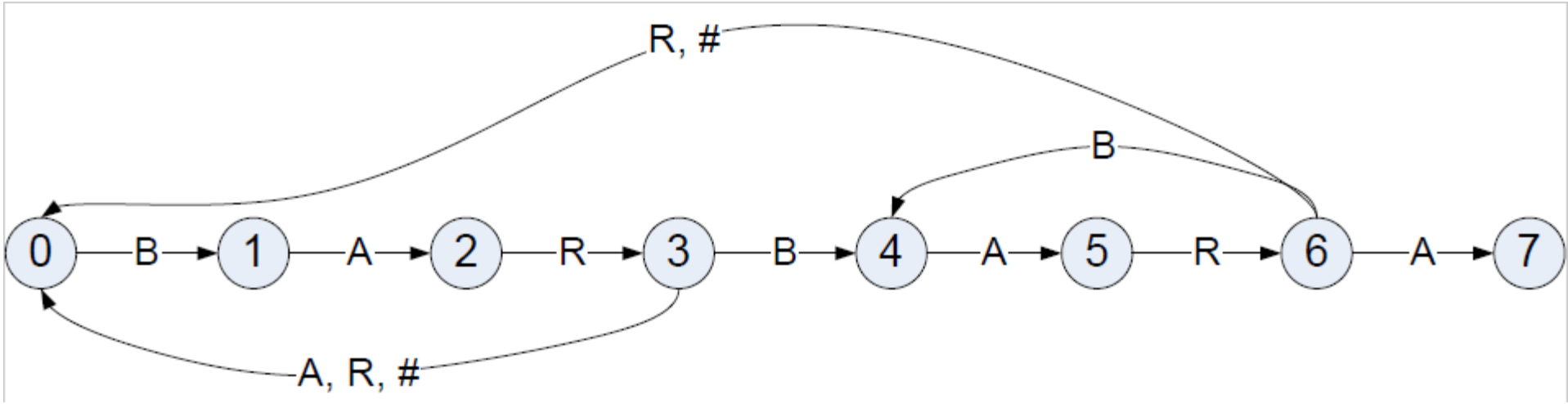


označava bilo koji simbol koji nije a ili n

Teži zadatak: konstruisati DFA koji prepoznaje reči koje sadrže patern “barbara”

Prefiksno stanje

- Neka simbol A prevodi stanje S_k u stanje S_{k+1}
- Stanje S_p ,p < k, je prefiksno stanje za stanje S_k ako za svaki simbol B, B ≠ A, važi dfa[B][S_k] = dfa[B][S_p] i p je najveće moguće
- **Primer. Kod automata za patern "BARBARA" stanje 3 (stanje u kome je prepoznato "BAR") je prefiksno za stanje 6 (stanje u kom je prepoznato "BARBAR")**
  - **Prefiksno stanje opisuje najveći pravi prefiks jednak sufiksu**



Konstrukcija DFA za proizvoljan patern

```
dfa[P[0]][0] = 1;
prefixState = 0;
for (int i = 1; i < P.length(); i++) {
      char c = P[i];
      dfa[c][i] = i + 1;
      for each symbol s in P different than c:
            dfa[s][i] = dfa[s][prefixState]

      // ažuriraj prefiksno stanje
      prefixState = dfa[c][prefixState]
}
dfa final state = P.length()
```

```java
public class DFASearch {
	private String pattern;
	private int[][] dfa;

	public DFASearch(String pattern) {
		this.pattern = pattern;
		buildDFA();
	}

	private void buildDFA() {
		HashSet<Character> chars = new HashSet<Character>();
		for (int i = 0; i < pattern.length(); i++)
			chars.add(pattern.charAt(i));

		dfa = new int[Character.MAX_VALUE][pattern.length() + 1];
		dfa[pattern.charAt(0)][0] = 1;
		int prefixState = 0;

		for (int i = 1; i < pattern.length(); i++) {
			int c = pattern.charAt(i);
			dfa[c][i] = i + 1;

			Iterator<Character> it = chars.iterator();
			while (it.hasNext()) {
				char ch = it.next();
				if (ch != c) dfa[ch][i] = dfa[ch][prefixState];
			}

			prefixState = dfa[c][prefixState];
		}
	}
```

```
public int searchIn(String text) {
	if (pattern.length() > text.length())
		throw new IllegalArgumentException("Patern veci od teksta");

	int state = 0;
	int finalState = pattern.length();

	for (int i = 0; i < text.length(); i++) {
		state = dfa[text.charAt(i)][state];
		if (state == finalState) {
			return i - pattern.length() + 1;
		}
	}

	return -1;
}
```

Brzo pretraživanje (Qucik search)

- Neka je dato neko poravnanje paterna (P) sa tekstom (T)
- **q** - probni znak - prvi znak T nakon poravnanja
- Poredimo karaktere P sa karakterima iz T u tekućem poravnanju grubom silom
- Kada dođe do neslaganja karaktera u prethodnom koraku tada
  1. Pomerimo P da se **q** iz T poklopi sa poslednjom pojavom **q** u P
  2. Ako se **q** ne pojavljuje u P tada P pomerimo iza **q.** U ovom slučaju pomeraj je jednak dužini P + 1.

Slučaj 1

m	a	r	m	a	r	m	i	r	k	o
m	i	r	k	o						
			m	i	r	k	o			

Slučaj 2

m	a	r	m	a	b	m	i	r	k	o
m	i	r	k	o						
						m	i	r	k	o

Slučaj 1 – mapa pomeraja

- Ukoliko se q pojavljuje u P tada P treba pomeriti za
  - 1 ukoliko je q poslednji karakter P
  - 2 ukoliko je q pretposlednji karakter P
  - 3 ukoliko je q pretpretposlednjui karakter P

    ...

- Formiraćemo mapu pomeraja u kojoj je ključ neki karakter iz P, a vrednost dužina pomeraja P za taj karakter.
  - Ova mapa se jednostavno formira obilaskom stringa od kraja ka početku
  - Pri tome treba voditi računa da u P neki znak može da se pojavi više puta – **treba nam poslednja pojava**

```
public class QuickSearch {
	private String pat;

	// slucaj 2: largeShift je duzina pomeraja iza probnog znaka
	private int largeShift;

	// slucaj 1: mapa pomeraja
	private HashMap<Character, Integer> shiftMap =
		new HashMap<Character, Integer>();

	public QuickSearch(String pat) {
		this.pat = pat;
		largeShift = pat.length() + 1;

		int shift = 1;
		for (int i = pat.length() - 1; i >= 0; i--) {
			char c = pat.charAt(i);
			if (!shiftMap.containsKey(c))
				shiftMap.put(c, shift);
			shift++;
		}
	}

	private int shift(char c) {
		Integer s = shiftMap.get(c);
		return s == null ? largeShift : s;
	}
	...
```

```
public int searchIn(String txt) {
	if (pat.length() > txt.length())
		throw new IllegalArgumentException("Patern veci od teksta");

	int i = 0; // pomeraj
	do {
		boolean match = true;
		int j = 0;
		while (match && j < pat.length()) {
			if (pat.charAt(j) != txt.charAt(i + j))
				match = false;
			else
				j++;
		}

		if (match) {
			return i;
		} else {
			// da li je tekuce poravnanje poslednje poravnanje?
			if (i == txt.length() - pat.length()) {
				return -1;
			}

			char test = txt.charAt(i + pat.length());
			i += shift(test);
		}
	} while (i <= txt.length() - pat.length());

	return -1;
}
```