Algoritmi in podatkovne strukture 2

Iskanje v nizih

Luka Fürst

Niz

- Končno zaporedje simbolov iz končne neprazne abecede Σ
- $S = a_1 a_2 \ldots a_n$
  - $a_i \in \Sigma$ za $i \in \{1, \ldots, n\}$
- $|S| = n$
  - dolžina niza S
- $S[i] = a_i$
  - indeksi se pričnejo z 1
  - se opravičujem, a tokrat mi to dejansko ustreza
- $S[i : j] = a_i a_{i+1} \ldots a_j$
- $S[i :] = a_i a_{i+1} \ldots a_n$
- $S[: i] = a_1 a_2 \ldots a_i$

} Odmislite python!

Niz

- Niz R je predpona niza S, če je $|R| \leq |S|$ in $R = S[: |R|]$
  - $R \sqsubseteq S$
  - $\texttt{aba} \sqsubseteq \texttt{aba} \sqsubseteq \texttt{abac}$
- Niz R je prava predpona niza S, če je $|R| < |S|$ in $R \sqsubseteq S$
  - $R \sqsubset S$
  - $\texttt{aba} \sqsubset \texttt{abac}$
- Niz R je pripona niza S, če je $|R| \leq |S|$ in $R = S[|S| - |R| + 1 :]$
  - $R \sqsupseteq S$
  - $\texttt{aba} \sqsupseteq \texttt{aba} \sqsupseteq \texttt{caba}$
- Niz R je prava pripona niza S, če je $|R| > |S|$ in $R \sqsupseteq S$
  - $R \sqsupset S$
  - $\texttt{aba} \sqsupset \texttt{caba}$

Iskanje vzorca v besedilu

- Naj bosta T in P niza nad abecedo Σ
- T: besedilo (*text*)
- P: vzorec (*pattern*)
- $|T| = n$
- $|P| = m$
- $1 \leq m \leq n$

Opis problema

- Poišči vse pojavitve niza P v nizu T
- Določi vse odmike s, tako da je $T[s+1 : s+m] = P$
  - $P \sqsupseteq T[s+m]$
- Primer
  - $T =$ `abbababacaba`
  - $P =$ `aba`
  - $s_1 = 3$ (`abbababacaba`)
  - $s_2 = 5$ (`abbababacaba`)
  - $s_3 = 9$ (`abbababacaba`)

Naivni pristop

- Za vsak $s \in \{0, 1, \ldots, n - m + 1\}$ preveri, ali je $T[s + 1 : s + m] = P$
- $n - m + 1$ odmikov (položajev vzorca v besedilu)
- $O(m)$ za primerjavo nizov $T[s + 1 : s + m]$ in P
- Skupaj $O(m(n - m + 1))$ ali (enostavneje) $O(mn)$
- Se da bolje?
- Asimpotična spodnja meja je namreč $\Omega(n)$

Rabin-Karpov algoritem

- Naj bo $\Sigma = \{a_1, a_2, \ldots, a_B\}$
- Niz nad Σ si lahko predstavljamo kot število v sistemu z osnovo B
- Naj bo $f(S)$ število, ki pripada nizu S
- $f(a_i) = i - 1$
- $f(a_{i_1} a_{i_2} \ldots a_{i_n}) = (i_1 - 1)B^{n-1} + (i_2 - 1)B^{n-2} + \ldots + (i_n - 1)B^0$

Primer

- $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$
- $B = 3$
- $f(\mathtt{a}) = 0,\ f(\mathtt{b}) = 1,\ f(\mathtt{c}) = 2$
- $f(\mathtt{babca}) = 10120_{(3)} = 1 \cdot 3^4 + 0 \cdot 3^3 + 1 \cdot 3^2 + 2 \cdot 3^1 + 0 \cdot 3^0 = 96$

Rabin-Karpov algoritem

- Funkcija f je pri fiksni dolžini niza bijektivna
  - če dolžina niza ni fiksna, to ne velja
  - $f(\texttt{ababca}) = f(\texttt{babca})$

- Vsi podnizi besedila T, ki se ujemajo z vzorcem P, so dolgi $|P|$, zato lahko upoštevamo bijektivnost in namesto P in $T[s+1:s+m]$ primerjamo $f(P)$ in $f(T[s+1:s+m])$

$$P = T[s+1:s+m] \iff f(P) = f(T[s+1:s+m])$$

Rabin-Karpov algoritem

- Potence B^0, B^1, ..., B^{m-1} izračunamo vnaprej
  - $O(m)$, enkratna operacija
- Za vzorec P izračunamo $f(P)$
  - $O(m)$, enkratna operacija
- Za vsak odmik s izračunamo $f(T[s+1:s+m])$
  - $O(m)$ za vsak odmik
  - $O(n-m+1)$ odmikov
  - skupaj $O((n-m+1)m)$
  - hm, to ni nič bolje kot naivno ...

Učinkovito posodabljanje $f(.)$

- Recimo, da smo v besedilu pravkar odkrili niz babca
- $f(\texttt{babca}) = f(\texttt{b}) \cdot 3^4 + f(\texttt{a}) \cdot 3^3 + f(\texttt{b}) \cdot 3^2 + f(\texttt{c}) \cdot 3^1 + f(\texttt{a})$
- Recimo, da je naslednji znak besedila enak c
- Ali lahko učinkovito izračunamo $f(\texttt{abcac})$ na podlagi $f(\texttt{babca})$?

Učinkovito posodabljanje $f(.)$

- Da!
- Naj bo $F = f(\texttt{babca})$
- Odbijemo prvi b
  - $F' = F - f(\texttt{b}) \cdot 3^4$
- Preostanek (abca) premaknemo za eno mesto v levo
  - $F'' = 3F'$
- Prištejemo $f(\texttt{c})$
- $f(\texttt{abcac}) = 3\left(f(\texttt{babca}) - f(\texttt{b}) \cdot 3^4\right) + f(\texttt{c})$

Učinkovito posodabljanje $f(.)$

- V splošnem:

$$f(T[s+2:s+m+1]) =$$
$$B\left(f(T[s+1:s+m]) - f(T[s+1])B^{m-1}\right) + f(T[s+m+1])$$

- $O(m)$ časa za enkratno predobdelavo vzorca
- $O(n-m+1)$ ali (enostavneje) $O(n)$ časa za iskanje vzorca v besedilu

Kaj pa, če . . .

- . . . sta abeceda in/ali vzorec P prevelika, da bi lahko $f(P)$ učinkovito izračunali?
- Spomnimo se na množenje velikih števil!

Računanje $f(.)$ po modulu

- V tem primeru nimamo druge možnosti, kot da $f(.)$ računamo po nekem modulu q
- Modul naj bo praštevilo in naj bo karseda velik
- Kljub temu se situacijam, ko je $f(S) = f(R)$ za $S \neq R$, v splošnem ne moremo izogniti
- Nizov dolžine m nad abecedo velikosti B je B^m, to pa je v realnih primerih praviloma bistveno več od smiselnega modula

Računanje $f(.)$ po modulu

- Če je $f(T[s+1:s+m]) \neq f(P)$, potem se P gotovo ne nahaja na odmiku s
- Če je $f(T[s+1:s+m]) = f(P)$, potem se P »zelo verjetno« nahaja na odmiku s, a moramo še preveriti, ali je res $T[s+1:s+m] = P$
- Na ta način z $O(n)$ ponovno pridemo na $O(mn)$, kljub temu pa je v praksi Rabin-Karp hitrejši od naivnega algoritma

Rabin-Karpov algoritem

function RABIN-KARP(T, P, B, q)
    $n \leftarrow |T|$
    $m \leftarrow |P|$
    $f \leftarrow 0$
    $g \leftarrow 0$
    $h \leftarrow B^{m-1} \bmod q$
    for $i \leftarrow 1$ **to** m **do**
        $f \leftarrow (Bf + P[i]) \bmod q$
        $g \leftarrow (Bg + T[i]) \bmod q$
    for $s \leftarrow 0$ **to** $n - m$ **do**
        if $f = g$ **then**
            if $P = T[s + 1 : s + m]$ **then**
                PRINT(s)
        if $s < n - m$ **then**
            $g \leftarrow (B(g - T[s + 1]h) + T[s + m + 1]) \bmod q$

Vrnimo se na osnovno metodo . . .

- Kako bi lahko povečali učinkovitost osnovne metode?
- Naj bo $P =$ `abcd`
- Recimo, da se `abc` ujema z besedilom na odmiku s, `d` pa se ne ujema

  T: . . .`abcx`. . .
  P: `abcd`

- Ali je nujno, da v naslednjem koraku poskusimo z odmikom $s + 1$?

Ključna ideja za izboljšavo osnovne metode

- Ne!
- Odmik s lahko povečamo za 4

  T: `...abcx.....` ==> `...abcx.....`
  P: `abcd` `abcd`

- Če bi se del `ab` ujemal z besedilom, `c` pa ne, bi odmik povečali za 3

  T: `...abx.....` ==> `...abx.....`
  P: `abcd` `abcd`

- Ali je povečanje odmika torej kar enako indeksu neujemanja (oz. m, če je $P = T[s+1 : s+m]$)?

Ključna ideja za izboljšavo osnovne metode

- Ne, žal ni tako enostavno
- Maksimalno varno povečanje odmika ni odvisno samo od položaja neujemanja, ampak tudi od znaka besedila na točki neujemanja in od strukture vzorca
- Analizirajmo dva primera ...

Ključna ideja za izboljšavo osnovne metode

- Naj bo r indeks prvega znaka v P, ki se pri trenutnem odmiku s razlikuje od istoležnega znaka v T
- $T[s+1:s+r] = P[1:r]$
- $T[s+r+1] \neq P[r+1]$
- $r = 0$, če imamo razliko že pri prvem znaku
- $r = m$, če imamo ujemanje med P in $T[s+1:s+m]$

- Za koliko lahko povečamo odmik (Δs) v odvisnosti od P, r in $T[s+r+1]$?

Ključna ideja za izboljšavo osnovne metode

- Naj bo $P = \texttt{abcd}$

r	$T[s+r+1]$	Δs
0	$\Sigma \setminus \{\texttt{a}\}$	1
1	$\Sigma \setminus \{\texttt{a}, \texttt{b}\}$	2
1	$\texttt{a}$	1
2	$\Sigma \setminus \{\texttt{a}, \texttt{c}\}$	3
2	$\texttt{a}$	2
3	$\Sigma \setminus \{\texttt{a}, \texttt{d}\}$	4
3	$\texttt{a}$	3
4	Σ	4

Ključna ideja za izboljšavo osnovne metode

- Naj bo $P = \mathtt{abac}$

r	$T[s+r+1]$	Δs
0	$\Sigma \setminus \{\mathtt{a}\}$	1
1	$\Sigma \setminus \{\mathtt{a}, \mathtt{b}\}$	2
1	$\mathtt{a}$	1
2	$\Sigma \setminus \{\mathtt{a}\}$	3
3	$\Sigma \setminus \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$	4
3	$\mathtt{a}$	3
3	$\mathtt{b}$	2
4	Σ	4

Ključna ideja za izboljšavo osnovne metode

- Naj bo $P =$ abac
- Recimo, da je $T[s+1 : s+4] =$ abab

T: ...abab..... ==> ...abab.....
P: abac abac

- Odmik lahko povečamo samo za 2, ker obstaja možnost, da je $T[s+3 : s+4] =$ ab začetek naslednje pojavitve P
- ab je pripona $T[: s+4]$ in hkrati predpona P

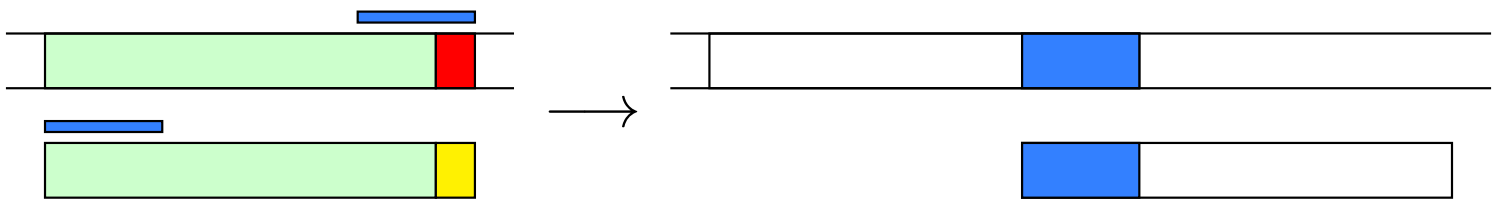
Priponska funkcija (glede na vzorec P)

- $\sigma_P(S)$ (ali kar $\sigma(S)$, če je P znan iz konteksta) je dolžina najdaljše predpone vzorca P, ki je hkrati pripona niza S
- $\sigma_P(S) = \max\{k \mid P[:k] \sqsupseteq S\}$
- Naj bo $P =$ `abac`

S	$\sigma(S)$
`xy`	0
`xya`	1
`xyab`	2
`xyaba`	3
`xyabac`	4

Priponska funkcija in največje varno povečanje odmika

- Naj bo $T[s+1:s+r] = P[1:r]$ in $T[s+r+1] \neq P[r+1]$
  - $r = 0$, če imamo razliko že pri prvem znaku
  - $r = m$, če imamo ujemanje med P in $T[s+1:s+m]$
- Naj bo $g = \sigma(T[:s+r+1])$
- Potem je $\Delta s = r + 1 - g$
- Ker je $T[s+r+2-g:s+r+1] = P[1:g]$ (po definiciji funkcije σ), v naslednjem koraku primerjamo znaka $T[s+r+2]$ in $P[g+1]$



Priponska funkcija in največje varno povečanje odmika

```
T:  ...abab.......    ==> ...abab.......
P:     abac                     abac
```

- $T[s+1:s+3] = P[1:3]$
- $T[s+4] \neq P[4]$
- $r = 3$
- $g = \sigma(T[:s+4]) = 2$
- $\Delta s = r + 1 - g = 3 + 1 - 2 = 2$
- V naslednjem koraku primerjamo znaka $T[s+r+2] = T[s+5]$ in $P[g+1] = P[3]$

Nekoliko drugačen pogled

- Lahko bi pri vsakem neujemanju računali funkcijo σ, a tako bi vnovič prišli do $O(mn)$, lahko pa ...
- ... upoštevamo, da je maksimalni varen Δs odvisen samo od strukture vzorca in od znaka neujemanja besedila, ne pa od celotnega besedila
- Zadošča torej, da izračunamo Δs za vsak par (položaj neujemanja, znak neujemanja)
- Če trenutni položaj v vzorcu obravnavamo kot stanje, povečanje odmika pa kot spremembo stanja, celotna zgodba zadiši (zasmrdi?) po ...

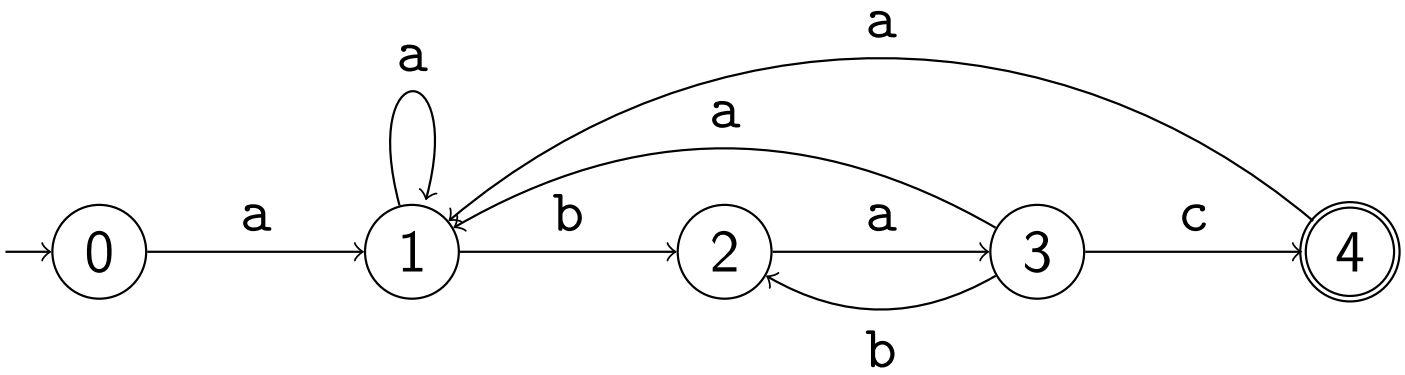
... deterministič̌nem končnem avtomatu, kakopak!

- $Q = \{0, 1, \ldots, m\}$
- $q_0 = 0$
- $F = \{m\}$
- $\delta(q, a) = \sigma(P[: q]a)$

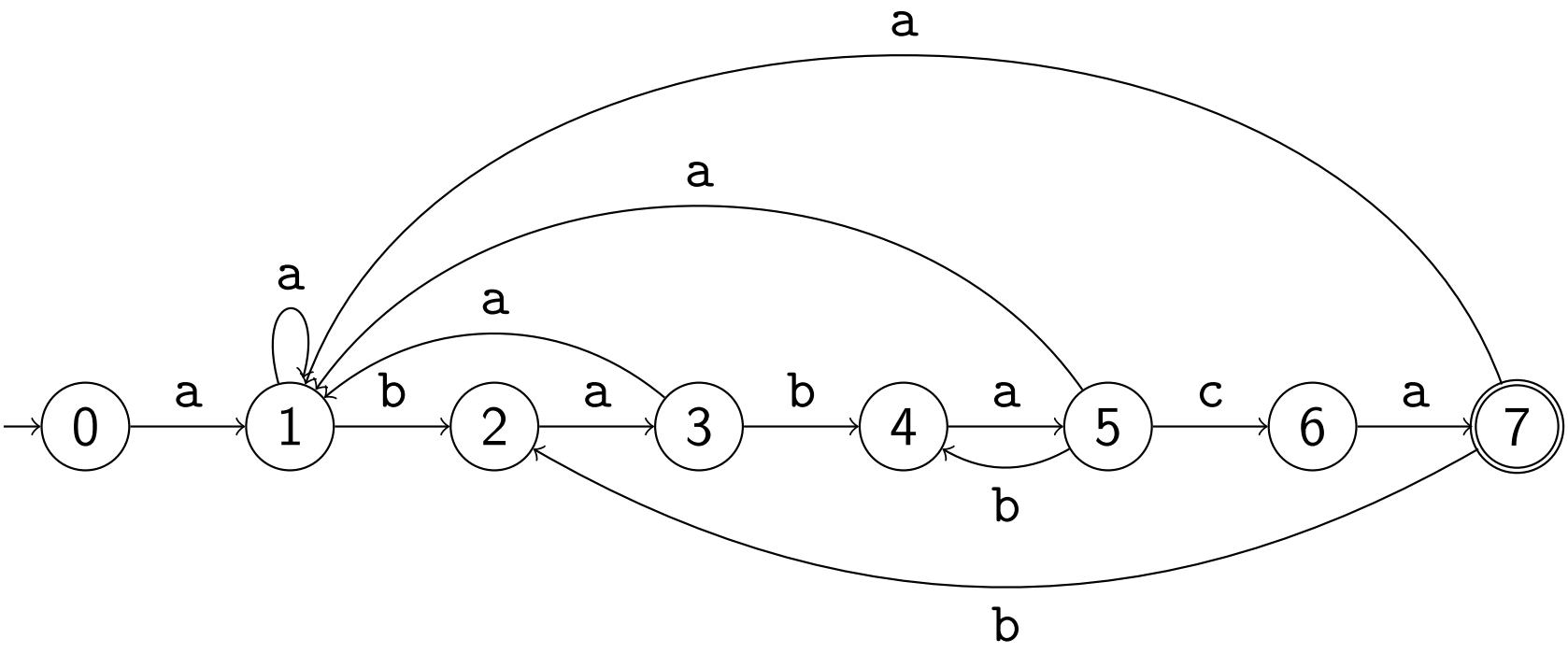
- Stanje nam pove število ujemajočih znakov besedila in vzorca
- Pričnemo v stanju 0, beremo znake besedila in potujemo po avtomatu
- Kadarkoli prispemo v končno stanje, najdemo pojavitev vzorca v besedilu

Primer 1: $P = \mathtt{abac}$

q	$\delta(q, \mathtt{a})$	$\delta(q, \mathtt{b})$	$\delta(q, \mathtt{c})$
0	1	0	0
1	1	2	0
2	3	0	0
3	1	2	4
4	1	0	0



Primer 2: $P =$ ababaca



Potovanje po DKA

- Recimo, da smo v stanju q
- To pomeni, da velja $T[s+1 : s+q] = P[: q]$
  - prvih q znakov vzorca se ujema z besedilom
- Naj bo $T[s+q+1] = a$
- Če je tudi $P[q+1] = a$, preidemo v stanje $q+1$
  - sedaj se prvih $q+1$ znakov vzorca ujema z besedilom
- Če je $P[q+1] \neq a$, preidemo v stanje $\sigma(P[: q]a) = q+1-\Delta s$, kjer je Δs največje varno povečanje odmika

Časovna zahtevnost

- Sprehod po besedilu zahteva le še $O(n)$ časa
- Gradnja avtomata je razmeroma zahtevna, a jo izvršimo samo enkrat
- $O(m^3|\Sigma|)$ po naivnem postopku
- Spodnja meja je $\Omega(m|\Sigma|)$
  - izračunati moramo $\delta(q, a)$ za vsako od $(m + 1)$ stanj in za vsakega od $|\Sigma|$ znakov abecede
- Lahko kaj prihranimo tudi v fazi preodbdelave?

Knuth-Morris-Prattov algoritem (KMP)

- Osnovna ideja: pri računanju maksimalnega varnega premika (= povečanja odmika) vzorca vzdolž besedila upoštevaj samo lastnosti vzorca in položaj neujemajočega znaka v vzorcu, ne pa tudi pripadajoči neujemajoči znak besedila
- Na ta način bo premik v nekaterih primerih morda manjši, kot bi lahko bil, zato pa bomo za preobdelavo vzorca potrebovali le $O(m)$ časa
- No, v resnici ne bodo premiki nič manjši kot pri iskanju z avtomatom, a do tja bomo še prišli ...

Največji varen premik

- Pri iskanju z avtomatom smo upoštevali položaj neujemanja in neujemajoči znak v besedilu

T: ...ab$x$.....
P: abcd

  - Pri $x = \mathsf{a}$ je $\Delta s = 2$, pri $x \in \Sigma \setminus \{\mathsf{a}, \mathsf{c}\}$ pa je $\Delta s = 3$

- Če neujemajočega znaka besedila ne upoštevamo, moramo predpostaviti najmanj ugodno možnost

T: ...ab?.....
P: abcd

  - Vzamemo $\Delta s = 2$ ne glede na znak ?

Največji varen premik pri vzorcu abcdefg

Položaj neujemanja	Situacija	Premik
1	...?..... abcdefg	1
2	...a?.... abcdefg	1
3	...ab?... abcdefg	2
4	...abc?.. abcdefg	3
5	...abcd?. abcdefg	4
6	...abcde? abcdefg	5
7	...abcdef? abcdefg	6
8	...abcdefg abcdefg	7

Največji varen premik pri vzorcu ababaca

Položaj neujemanja	Situacija	Premik
1	...?..... ababaca	1
2	...a?.... ababaca	1
3	...ab?... ababaca	2
4	...aba?.. ababaca	2
5	...abab?. ababaca	2
6	...ababa? ababaca	2
7	...ababac ababaca	6
8	...ababaca ababaca	6

Formula za največji varen premik

- i: indeks zadnjega ujemajočega znaka v besedilu
- j: število ujemajočih znakov med besedilom in vzorcem
- $\pi[j]$: maksimalni $k < j$, tako da je $T[i-k+1:i] = P[1:k]$

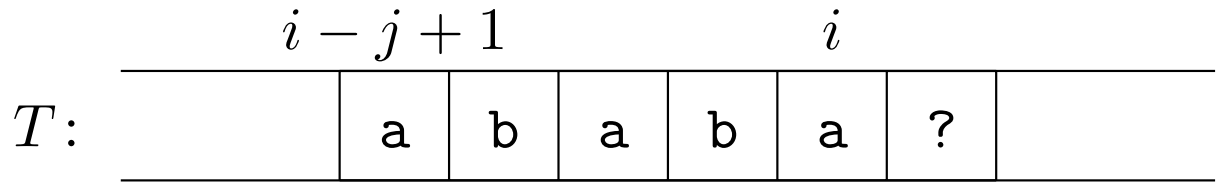
  T: ...ababa?....
  P: ababaca

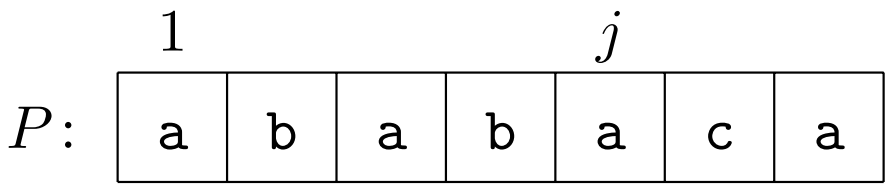
  - $j = 5$
  - $\pi[j] = 3$

- Pri $j = 0$ je $\Delta s = 1$
- Sicer je $\Delta s = j - \pi[j]$

Predponska funkcija

- $\pi[j] = \max\{k < j \mid T[i-k+1:i] = P[1:k]\}$
- Ker je j število ujemajočih znakov med T in P, i pa indeks zadnjega ujemajočega znaka v T, je $T[i-j+1:i] = P[1:j]$ in zato tudi $T[i-k+1:i] = P[j-k+1:j]$

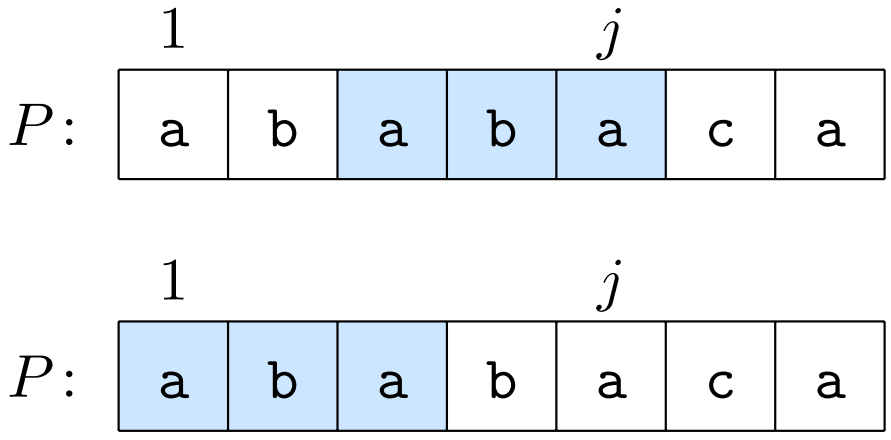




Predponska funkcija

- Predponsko funkcijo (π) lahko torej izračunamo zgolj na podlagi vzorca

$$\pi[j] = \max\{k < j \mid P[j-k+1 : j] = P[1 : k]\}$$



- Z drugimi besedami: $\pi[j]$ je dolžina najdaljše prave pripone niza $P[: j]$, ki je hkrati predpona niza P
- $\pi[j] = \max\{|R| \mid R \sqsupset P[: j] \wedge R \sqsubset P\}$

Uporaba predponske funkcije

- Vnaprej izračunamo $\pi[.]$ za vsak indeks v P

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

- Pojavitve P v T iščemo na običajen način (s pomikanjem P vzdolž T)
- Na podlagi vrednosti $\pi[j]$, kjer je j indeks zadnjega ujemajočega znaka v vzorcu, izračunamo največji varen premik vzorca vzdolž besedila (Δs)
- *Voilà*!

Izračun predponske funkcije

- Hm ...
- Koliko časa pa traja izračun predponske funkcije?
- $O(m^2)$ po definiciji!
- Knuth, Morris in Pratt se s tem niso zadovoljili ...

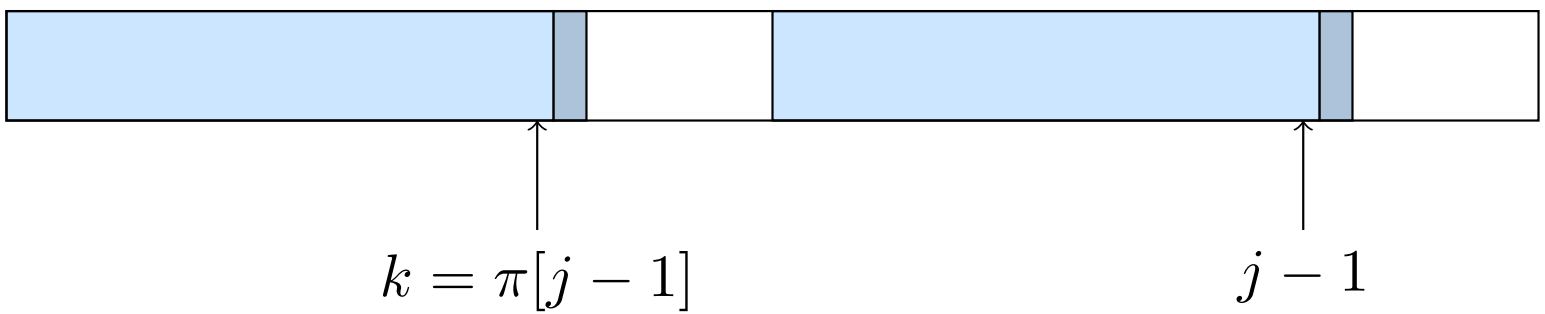
Učinkovitejše računanje $\pi[.]$

- Razmišljajmo induktivno ...
- $\pi[1]$ je kakopak vedno 0
- Recimo, da smo že izračunali $\pi[1]$, $\pi[2]$, ..., $\pi[j-1]$
- Kako izračunamo $\pi[j]$?

Primer 1

- Naj bo $k = \pi[j-1]$
- Po definiciji je $P[j-k : j-1] = P[1 : k]$
- Če je tudi $P[j] = P[k+1]$, lahko zaključimo

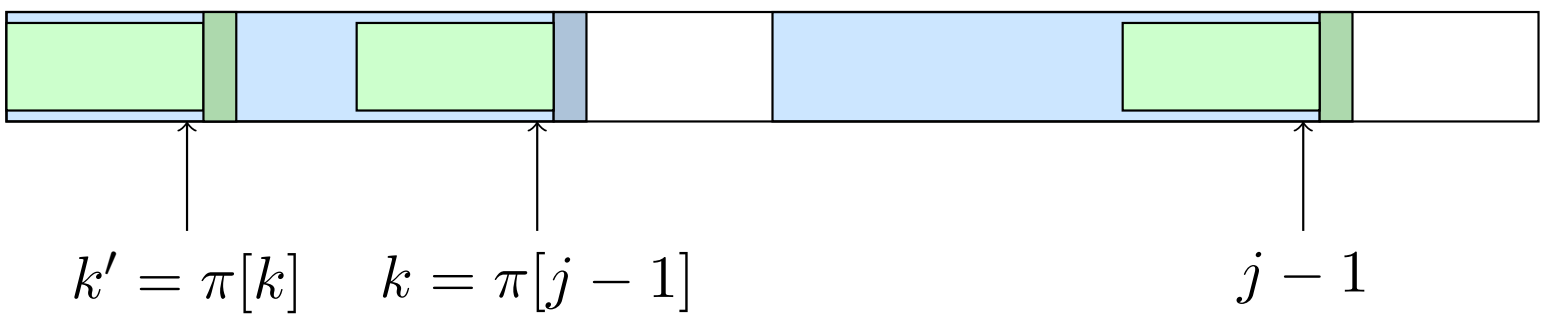
$$\pi[j] = k + 1 = \pi[j-1] + 1$$



Primer 2

- Kaj pa, če $P[j] \neq P[k+1]$?
- Naj bo $k' = \pi[k]$
- Velja $P[k-k'+1:k] = P[1:k']$
- Ker je $P[j-k:j-1] = P[1:k]$, je tudi $P[j-k':j-1] = P[k-k'+1:k]$
- Iz prejšnjih dveh alinej sledi $P[j-k':j-1] = P[1:k']$
- Če je $P[j] = P[k'+1]$, lahko zaključimo

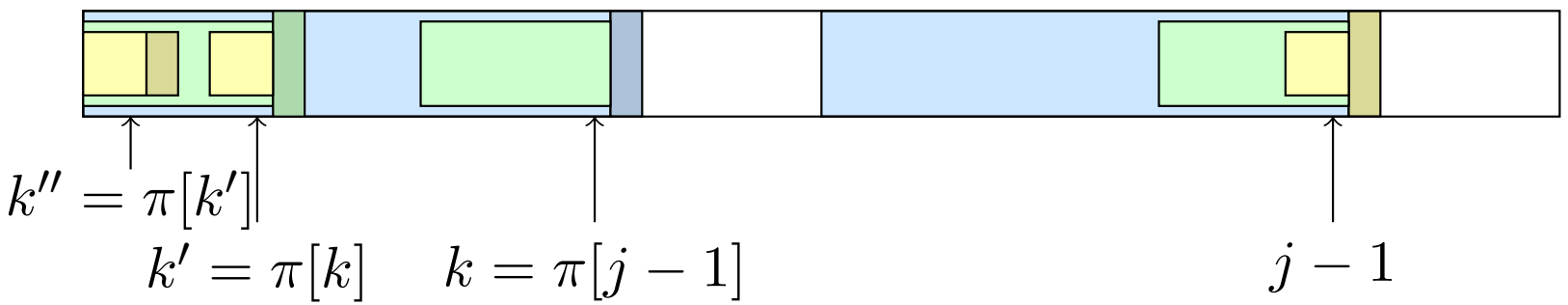
$$\pi[j] = k' + 1 = \pi[k] + 1 = \pi[\pi[j-1]] + 1$$



Primer 3

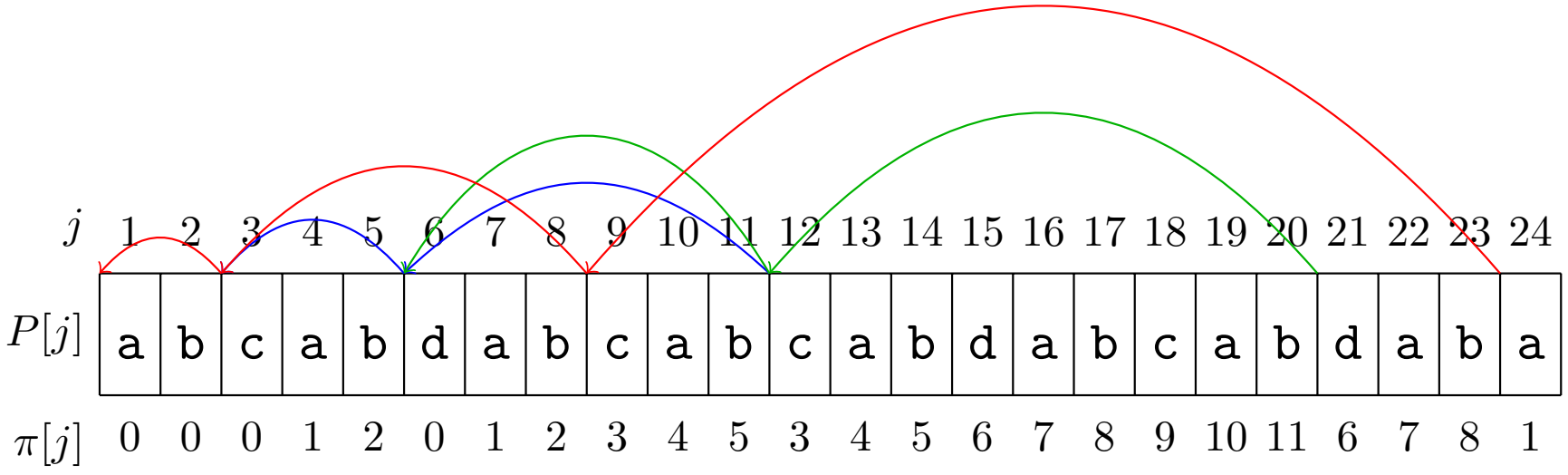
- Kaj pa, če je tudi $P[j] \neq P[k' + 1]$?
- Naj bo $k'' = \pi[k']$
- Po isti logiki kot prej velja $P[j - k'' : j - 1] = P[1 : k'']$
- Če je $P[j] = P[k'' + 1]$, lahko zaključimo

$$\pi[j] = k'' + 1 = \pi[\pi[\pi[j - 1]]] + 1$$



- In tako naprej ...

Konkreten primer



- Izračun $\pi[12]$
  - $k = \pi[11] = 5$
  - $T[7:11] = T[1:5]$, vendar pa $T[12] \neq T[6]$
  - $k' = \pi[k] = \pi[5] = 2$
  - $T[10:11] = T[1:2]$, poleg tega pa tudi $T[12] = T[3]$
  - Torej je $\pi[12] = \pi[5] + 1 = \pi[\pi[11]] + 1 = 3$

Računanje predponske funkcije

function IZRAČUNAJ-PREDPONSKO-FUNKCIJO(P)
    $m \leftarrow |P|$
    Izdelaj tabelo $\pi[1 : m]$
    $\pi[1] \leftarrow 0$
    $k \leftarrow 0$
    for $j \leftarrow 2$ **to** m **do**
        ▷ *Poišči največji k, da bo $P[j - k + 1 : j] = P[1 : k]$.* ◁
        while $k > 0 \wedge P[k + 1] \neq P[j]$ **do**
            $k \leftarrow \pi[k]$
        if $P[k + 1] = P[j]$ **then**
            $k \leftarrow k + 1$
        $\pi[j] \leftarrow k$
    return π

Časovna zahtevnost

- Na prvi pogled $O(m^2)$, a ta zgornja meja ni tesna
- k se poveča največ $(m-1)$-krat (vsakokrat za 1)
- k se v vsaki iteraciji zanke while zmanjša vsaj za 1 (ker je $\pi[i] < i$ za vsak i), a nikoli ne postane negativen
- Skupno število zmanjševanj k potemtakem ne more biti večje od skupnega števila povečevanj k (torej $m-1$)
- Skupno število operacij je potemtakem $O(m)$

Iskanje pojavitev P v T

- Z indeksom i oz. j vzporedno potujemo po T in P, dokler je $T[i] = P[j]$
- Ko naletimo na i in j, tako da je $T[i+1] \neq P[j+1]$, bi lahko P povsem varno premaknili za $j - \pi[j]$ mest naprej
- Z upoštevanjem znaka $T[i+1]$, ki je povzročil neujemanje, pa lahko dosežemo tudi daljše premike
- Iščemo najdaljšo pripono niza $T[: i+1]$, ki je enaka predponi niza P

Primer 1

```
T   . . . a b c a b d a b c a b d . . . . . . . . .
P         a b c a b d a b c a b e
π         0 0 0 1 2 0 1 2 3 4 5 0

T   . . . a b c a b d a b c a b d . . . . . . . . .
P                     a b c a b d a b c a b e
```

- V tem primeru je največji premik dejansko $j - \pi[j] = 11 - 5 = 6$

Primer 2

```
T  . . . a b c a b d a b c a b c . . . . . . . . . .
P        a b c a b d a b c a b e
π        0 0 0 1 2 0 1 2 3 4 5 0

T  . . . a b c a b d a b c a b c . . . . . . . . . .
P                    a b c a b d a b c a b e

T  . . . a b c a b d a b c a b c . . . . . . . . . .
P                          a b c a b d a b c a b e
```

- V tem primeru je premik $j - \pi[\pi[j]] = 11 - 2 = 9$

Primer 3

```
T  . . . a b c a b d a b c a b a . . . . . . . . . . . .
P        a b c a b d a b c a b e
π        0 0 0 1 2 0 1 2 3 4 5 0

T  . . . a b c a b d a b c a b a . . . . . . . . . . . .
P                    a b c a b d a b c a b e

T  . . . a b c a b d a b c a b a . . . . . . . . . . . .
P                          a b c a b d a b c a b e

T  . . . a b c a b d a b c a b a . . . . . . . . . . . .
P                               a b c a b d a b c a b e
```

- V tem primeru je premik $j - \pi[\pi[\pi[j]]] = 11$

Iskanje pojavitev P v T

- Naj bo $T[i-j+1:i] = P[1:j]$ in $T[i+1] \neq P[j+1]$
- Vzorec P premaknemo tako, da dosežemo $T[i-k+2:i+1] = P[1:k]$
  - $k = \pi[\pi[\ldots[\pi[j]]\ldots]] + 1$
- V naslednji iteraciji primerjamo $T[i+2]$ in $P[k+1]$, saj so vsi predhodni znaki vzorca enaki istoležnim znakom besedila
- k je torej dolžina najdaljše predpone vzorca, ki je hkrati pripona niza $T[:i+1]$
- $k = \sigma(T[:i+1]) = \sigma(P[:j]T[i+1])$, kjer je σ priponska funkcija, na kateri temelji iskanje s končnim avtomatom
- Iskanje s KMP je torej enakovredno iskanju z avtomatom

Iskanje pojavitev P v T

```
function KMP(T, P)
    π ← Izračunaj-predponsko-funkcijo(P)
    j ← 0
    for i ← 1 to n do
        while j > 0 ∧ P[j + 1] ≠ T[i] do
            j ← π[j]
        if P[j + 1] = T[i] then
            j ← j + 1
        if j = m then
            Print(i − m)                ▷ položaj pojavitve vzorca
            j ← π[j]
```

Časovna zahtevnost

- j se poveča kvečjemu n-krat (vsakokrat kvečjemu za 1)
- V vsaki iteraciji zanke while se j strogo zmanjša (ker je $\pi[j] < j$)
- Skupno število zmanjševanj j torej ne more biti večje od skupnega števila povečevanj j (tj. n)
- Časovna zahtevnost je potemtakem $O(n)$

Priponska tabela (*suffix array*)

- Knuth-Morris-Pratt, Z-algoritem in Boyer-Moore[1] poiščejo vzorec P dolžine m v besedilu T dolžine n v času $O(n)$, pri čemer potrebujejo $O(m)$ časa za obdelavo vzorca
- Skupen čas iskanja je tako $O(n + m)$
- Priponska tabela za besedilo T nam omogoča, da za poljuben vzorec P poiščemo vse njegove pojavitve v T v času $O(m \log n + km)$, kjer je k število pojavitev vzorca

[1] Še ena poslastica — gl. npr. D. Gusfield: *Algorithms on Strings, Trees, and Sequences*

Priponska tabela

- Kompakten zapis leksikografsko urejenega zaporedja vseh pripon besedila T
- $SA[i]$: indeks v nizu T, na katerem se prične pripona, ki je v leksikografsko urejenem zaporedju pripon na i-tem mestu
- Leksikografsko naraščajoče zaporedje pripon je potemtakem $T[SA[1]:], T[SA[2]:], \ldots, T[SA[n]:]$

Primer za $T =$ abrakadabra

i	$SA[i]$	$T[SA[i]:]$
1	11	a
2	8	abra
3	1	abrakadabra
4	6	adabra
5	4	akadabra
6	9	bra
7	2	brakadabra
8	7	dabra
9	5	kadabra
10	10	ra
11	3	rakadabra

Iskanje s pomočjo priponske tabele

- Vzorec P se nahaja v besedilu T natanko tedaj, ko je P predpona neke pripone niza T
- $P = T[i : i + m - 1] \iff P \sqsubseteq T[i :]$
- Število pojavitev niza P je enako številu pripon niza T, katerih predpona je P
- Niz `br` se v nizu `abrakadabra` nahaja dvakrat, ker je `br` predpona dveh pripon (`bra` in `brakadabra`)
- Ker so pripone v priponski tabeli urejene leksikografsko, lahko po njih iščemo z bisekcijo

Iskanje s pomočjo priponske tabele

- Primerjava vzorca s pripono (preverjanje, ali je $P \sqsubseteq T[i:]$) terja $O(m)$ časa
- $O(m \log n)$, da odkrijemo prvo pojavitev vzorca
- Ko odkrijemo začetno pojavitev vzorca P, se po priponski tabeli sprehodimo še naprej in nazaj od tistega položaja, da odkrijemo še ostale pojavitve (indekse i', tako da je $P \sqsubseteq T[i':]$)
- $O(m \log n + km)$, da odkrijemo vseh k pojavitev vzorca

Naivna izdelava priponske tabele

- Uredimo tabelo $[1, 2, \ldots, n]$, pri čemer je $i \prec j$ natanko tedaj, ko je niz $T[i:]$ leksikografsko manjši od niza $T[j:]$
- Niza dolžine (največ) n med seboj primerjamo v času $O(n)$
- $O(n^2 \log n)$ za urejanje n nizov dolžine največ n

Učinkovitejši pristop

- Naj bosta x in y poljubna niza
- Naj bo $x = x_1 x_2$ in $y = y_1 y_2$ ($|x_1| = |y_1|$)
- Potem velja

$$x \prec y \iff x_1 \prec y_1 \vee (x_1 = y_1 \wedge x_2 \prec y_2)$$

- Pripone uredimo v $\lceil \lg n \rceil + 1$ iteracijah, pri čemer jih v ℓ-ti iteraciji uredimo glede na prvih $2^{\ell-1}$ znakov

Učinkovitejši pristop

- V ℓ-ti iteraciji dodelimo indeksu $i \in \{1, \ldots, n\}$ rang $r_\ell(i) \in \{1, \ldots, n\}$, tako da velja

$$r_\ell(i) < r_\ell(j) \iff T[i:][:2^{\ell-1}] \prec T[j:][:2^{\ell-1}]$$

- Range $r_{\ell+1}(.)$ dobimo iz rangov $r_\ell(.)$ tako, da indekse uredimo glede na pare $(r_\ell(i), r_\ell(i + 2^{\ell-1}))$
  - $r_\ell(i)$: rang niza $T[i:][:2^{\ell-1}]$
  - $r_\ell(i + 2^{\ell-1})$: rang niza $T[i + 2^{\ell-1}:][:2^{\ell-1}]$
  - $(r_\ell(i), r_\ell(i + 2^{\ell-1}))$: izhodišče za rang niza $T[i:][:2^\ell]$

- Neobstoječi podnizi imajo rang 0 '
  - prazen niz je leksikografsko manjši od vseh ostalih '

Primer: rangi v iteraciji 1

i	$T[i:][:1]$	$r_1(i)$
1	a	1
2	b	2
3	r	5
4	a	1
5	k	4
6	a	1
7	d	3
8	a	1
9	b	2
10	r	5
11	a	1

Prehod z iteracije 1 na iteracijo 2

i	$T[i:][:2]$	$(r_1(i), r_1(i+1))$	i	$T[i:][:2]$	$(r_1(i), r_1(i+1))$	$r_2(i)$
1	ab	$(1,2)$	11	a	$(1,0)$	1
2	br	$(2,5)$	1	ab	$(1,2)$	2
3	ra	$(5,1)$	8	ab	$(1,2)$	2
4	ak	$(1,4)$	6	ad	$(1,3)$	3
5	ka	$(4,1)$	4	ak	$(1,4)$	4
6	ad	$(1,3)$	2	br	$(2,5)$	5
7	da	$(3,1)$	9	br	$(2,5)$	5
8	ab	$(1,2)$	7	da	$(3,1)$	6
9	br	$(2,5)$	5	ka	$(4,1)$	7
10	ra	$(5,1)$	3	ra	$(5,1)$	8
11	a	$(1,0)$	10	ra	$(5,1)$	8

Rangi v iteraciji 2

i	$T[i:][:2]$	$r_2(i)$
1	`ab`	2
2	`br`	5
3	`ra`	8
4	`ak`	4
5	`ka`	7
6	`ad`	3
7	`da`	6
8	`ab`	2
9	`br`	5
10	`ra`	8
11	`a`	1

Prehod z iteracije 2 na iteracijo 3

i	$T[i:][:4]$	$(r_2(i),\, r_2(i+2))$	i	$T[i:][:4]$	$(r_2(i),\, r_2(i+2))$	r_3
1	`abra`	$(2,8)$	11	`a`	$(1,0)$	1
2	`brak`	$(5,4)$	1	`abra`	$(2,8)$	2
3	`raka`	$(8,7)$	8	`abra`	$(2,8)$	2
4	`akad`	$(4,3)$	6	`adab`	$(3,2)$	3
5	`kada`	$(7,6)$	4	`akad`	$(4,3)$	4
6	`adab`	$(3,2)$	9	`bra`	$(5,1)$	5
7	`dabr`	$(6,5)$	2	`brak`	$(5,4)$	6
8	`abra`	$(2,8)$	7	`dabr`	$(6,5)$	7
9	`bra`	$(5,1)$	5	`kada`	$(7,6)$	8
10	`ra`	$(8,0)$	10	`ra`	$(8,0)$	9
11	`a`	$(1,0)$	3	`raka`	$(8,7)$	10

Rangi v iteraciji 3

i	$T[i:][:4]$	$r_3(i)$
1	`abra`	2
2	`brak`	6
3	`raka`	10
4	`akad`	4
5	`kada`	8
6	`adab`	3
7	`dabr`	7
8	`abra`	2
9	`bra`	5
10	`ra`	9
11	`a`	1

Prehod z iteracije 3 na iteracijo 4

i	$T[i:][:8]$	$(r_3(i), r_3(i+4))$	i	$T[i:][:8]$	$(r_3(i), r_3(i+4))$	$r_4(i)$
1	`abrakada`	$(2, 8)$	11	`a`	$(1, 0)$	1
2	`brakadab`	$(6, 3)$	8	`abra`	$(2, 0)$	2
3	`rakadabr`	$(10, 7)$	1	`abrakada`	$(2, 8)$	3
4	`akadabra`	$(4, 2)$	6	`adabra`	$(3, 9)$	4
5	`kadabra`	$(8, 5)$	4	`akadabra`	$(4, 2)$	5
6	`adabra`	$(3, 9)$	9	`bra`	$(5, 0)$	6
7	`dabra`	$(7, 1)$	2	`brakadab`	$(6, 3)$	7
8	`abra`	$(2, 0)$	7	`dabra`	$(7, 1)$	8
9	`bra`	$(5, 0)$	5	`kadabra`	$(8, 5)$	9
10	`ra`	$(9, 0)$	10	`ra`	$(9, 0)$	10
11	`a`	$(1, 0)$	3	`rakadabr`	$(10, 7)$	11

Rangi v iteraciji 4

i	$T[i:][:8]$	$r_4(i)$
1	`abrakada`	3
2	`brakadab`	7
3	`rakadabr`	11
4	`akadabra`	5
5	`kadabra`	9
6	`adabra`	4
7	`dabra`	8
8	`abra`	2
9	`bra`	6
10	`ra`	10
11	`a`	1

Končni rangi in priponska tabela

- Ker so vsi rangi med seboj različni, nam iteracije 5 ni treba izvesti
- Iz končnih rangov zlahka pridobimo priponsko tabelo

$$r(i) = j \iff SA[j] = i$$

i	$T[i:]$	$r(i)$	$SA[i]$	$T[SA[i]:]$
1	abrakadabra	3	11	a
2	brakadabra	7	8	abra
3	rakadabra	11	1	abrakadabra
4	akadabra	5	6	adabra
5	kadabra	9	4	akadabra
6	adabra	4	9	bra
7	dabra	8	2	brakadabra
8	abra	2	7	dabra
9	bra	6	5	kadabra
10	ra	10	10	ra
11	a	1	3	rakadabra

Časovna zahtevnost

- $O(\log n)$ iteracij
- $O(n \log n)$ v vsaki iteraciji (urejanje rangov)
- Skupaj $O(n \log^2 n)$
- Ker so rangi v vsaki iteraciji števila iz množice $\{1, \ldots, n\}$, jih lahko uredimo s postopkom, ki teče v času $O(n)$
  - npr. korensko urejanje ali urejanje s štetjem
- Na ta način dobimo $O(n \log n)$