UNIVERZA V LJUBLJANI
FAKULTETA ZA MATEMATIKO IN FIZIKO
Matematika – praktična matematika (VSŠ)

Anja ROŽAC

Algoritmi za delo z nizi

Diplomska naloga

Ljubljana, 2009

DIPLOMSKA NALOGA :
FAKULTETA ZA MATEMATIKO IN FIZIKO

ZAHVALA

Mentorju mag. Matiji Lokar se iskreno zahvaljujem, da me je sprejel pod svoje mentorstvo in mi skrbno pregledoval osnutke diplomske naloge ter me usmerjal v pravo smer.
Posebna zahvala pripada družini in Davidu, ki so mi stali ob strani in me spodbujali.

DIPLOMSKA NALOGA :
FAKULTETA ZA MATEMATIKO IN FIZIKO

KAZALO

PROGRAM DELA

V diplomski nalogi predstavite osnovne sestavljene operacije, ki jih izvajamo nad nizi, kot so iskanje podniza v danem nizu, primerjava podobnosti dveh nizov, stiskanje nizov.

Osnovna literatura:

Goodrich, M. T., Tamassia, R.: *Data structures and algorithms in java*, J.Wiley & Sons, 2001, New York.

mentor

mag. Matija Lokar

DIPLOMSKA NALOGA :
FAKULTETA ZA MATEMATIKO IN FIZIKO

POVZETEK

Osnovne operacije, ki jih izvajamo nad besedilom, so iskanje besede, stavka v danem besedilu, primerjava dveh danih besedil, stiskanje besedila.
Cilj diplomske naloge je na enostaven, razumljiv način prikazati algoritme, s pomočjo katerih lahko izvajamo omenjene operacije. Diplomska naloga je razdeljena na tri glavna poglavja. V poglavju 1 so razloženi osnovni algoritmi, ki jih uporabljamo pri iskanju določenega podniza v danem besedilu. Poglavje 2 opisuje tehniko stiskanja teksta, ki jo imenujemo Huffmanovo kodiranje. V zadnjem poglavju, poglavju 3, je razložena metoda, ki jo uporabljamo pri preverjanju, ali sta dani besedili enaki.

Math. Subj. Class. (2000): 68Q25, 68Q30, 68W05, 68W40.
Computing Review Class. System (1998): E.4, E.5, F.2.2.
Ključne besede: *besedilo,vzorec, niz, podniz, pripona, predpona, algoritem*

ABSTRACT

Basic operations being done on a text are word and sentence search, text comparison and text compression.
The goal of this thesis is to present algorithms, which help us to execute operations mentioned above in a simple, understandable way. The thesis consists of three major chapters. In chapter 1 basic algorithms which are used for searching for substring in a given text are explained. Chapter 2 describes text compression technique, known as Huffman coding. The last chapter, chapter 3, describes a method used for text similarity testing.

Math. Subj. Class. (2000): 68Q25, 68Q30, 68W05, 68W40
Computing Review Class. System (1998): E.4, E.5, F.2.2.
Key words: *text, pattern, string, substring, suffix, preffix, algorithm*

1 Algoritmi za iskanje vzorcev v besedilu

Pri klasičnem problemu iskanja vzorcev v besedilu imamo podano besedilo T dolžine n in vzorec P dolžine m. Zanima nas, ali je vzorec P podniz besedila T, ali kot tudi rečemo, ali se vzorec P ujema z besedilom T. Ujemanje obstaja, ko v besedilu T najdemo podniz, ki se začne na nekem indeksu i v T in se znak po znak ujema z vzorcem P. Veljati mora torej

T[i] = P[0], T[i + 1] = P[1], …, T[i + m - 1] = P[m - 1].

To lahko zapišemo tudi kot P = T[i … i + m - 1]. Tako se na primer vzorec "anja" ujema z besedilom "Tanja", vzorec "ble" z besedilom "To je blebetanje, blebetanje in nič drugega". Vzorec "matematika" se z besedilom "Fakulteta za matematiko in fiziko" ne ujema.
Večina algoritmov za iskanje vzorcev v besedilu kot rezultat vrne število. To je običajno bodisi -1, kar pomeni, da vzorec P ne obstaja v besedilu T, bodisi pozitivno celo število, ki označuje začetni indeks ujemanja podniza v besedilu T.

Oglejmo si primer. Podano imamo besedilo T = "algoritmi za iskanje" in vzorec P = "iskan".
Najprej zapišimo oba niza vsakega v svojo tabelo.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
T[i]	a	l	g	o	r	i	t	m	i		z	a		i	s	k	a	n	j	e

j	0	1	2	3	4
P[j]	i	s	k	a	n

Dolžina besedila T je 20 (dožino besedila bomo običajno označevali z n), dolžina vzorca P pa je 5 (dolžino vzorca bomo označevali z m). Že na prvi pogled opazimo ujemanje vzorca P s podnizom od T[13] do T[17]. Torej velja T[13] = P[0], T[14] = P[1], …, T[17] = P[4]. Algoritmi bi vrnili število 13, ki predstavlja začetni indeks ujemanja vzorca P v danem besedilu T.

V primeru, ko je dolžina vzorca P večja od dolžine besedila T, ali, ko v T-ju ne najdemo vzorca P, ujemanja ni. Algoritmi takrat vrnejo -1. Pri večini algoritmov pri preverjanju ujemanja ločimo male črke od velikih, kar pomeni, da 'a' ≠ 'A'.

Poznamo več algoritmov za iskanje vzorcev v besedilu. V diplomski nalogi bomo predstavili tri. To so Groba Sila (Brute Force), algoritem Boyer-Moore in algoritem Knuth-Morris-Pratt. Algoritem Groba Sila temelji na zaporednih primerjanjih vzorca z besedilom in je časovno precej zahteven. Značilnost algoritma Boyer-Moore je, da se iskanje vzorca začne od zadnje črke v vzorcu in gre proti prvi črki v vzorcu. Algoritem Knuth-Moriss-Pratt tako kot algoritem Groba sila uporablja zaporedno iskanje, le da je zaradi posebne metode iskanja hitrejši.

Algoritme bomo zapisali v programskem jeziku java. Napisali bomo razred `TekstovniAlgoritmi`, ki bo vseboval tri metode, in sicer `algoritemGrobaSila`, `algoritemBoyerMoore` in `algoritemKnuthMorrisPratt`. Metode bodo kot vhodna parametra dobile spremenljivki `p` in `t`, ki bosta tipa `String`. Kot rezultat bodo vrnile celo število, ki bo označevalo začetni indeks ujemanja vzorca P v besedilu T. V primeru, da ujemanja ni, bodo vrnile `-1`. V razredu `TekstovniAlgoritmi` bodo implementirane tudi nekatere privatne metode, ki jih bomo potrebovali pri gradnji algoritmov.

```
import java.util.*;
public class TekstovniAlgoritmi
{

  public static int algoritemGrobaSila(String t, String p)
  {
    //implementacija algoritma Groba Sila
  }

  public static int algoritemBoyerMoore(String t, String p)
  {
    //implementacija algoritma Boyer-Moore
  }

  public static int algoritemKnuthMorrisPratt(String t, String p)
  {
    //implementacija algoritma Knuth-Morris-Pratt
  }

  //tu bodo implementirane tudi privatne metode, ki jih bomo
  //potrebovali
}
```

Za prikaz uporabe algoritmov bomo napisali poseben razred `Uporaba`, ki bo v metodi `main` klical zgoraj omenjene algoritme. V tej metodi bomo predpostavili, da imamo besedilo T shranjeno v spremenljivki `t` tipa `String`. V tem besedilu bomo iskali vzorec P, ki bo shranjen v spremenljivki `p` tipa `String`.
Oglejmo si program za prikaz uporabe algoritmov, zapisan v javanski kodi.

```
import java.util.*;

public class Uporaba
{
  public static void main(String[] args)
  {
    //dano besedilo
    String t = "Algoritmi za iskanje vzorcev v besedilu";
    //iskani vzorec v besedilu
    String p = "iskanje";

    //pokličemo algoritem Groba Sila, s katerim preverjamo,
    // ali vzorec p obstaja v besedilu t
    int mestoUjemanja = TekstovniAlgoritmi.algoritemGrobaSila(t, p);

    //če algoritem vrne pozitivno celo število pomeni,
    // da smo našli začetni indeks ujemanja vzorca p v besedilu t
    if(mestoUjemanja != -1)
    {
      System.out.println("Ujemanje se začne na mestu "
        + mestoUjemanja);
    }
    else
    {
      //če algoritem vrne -1 izpišemo prijazno besedilo, s katerim
      //povemo, da v besedilu t ne obstaja podniz, ki bi se v
      //celoti ujemal z iskalnim vzorcem p
      System.out.println("Ujemanja nismo našli.");
   }
  }
}
```

V naslednjih razdelkih bomo podrobneje predstavili omenjene tri algoritme.

1.1 Groba Sila (Brute Force)

Ideja tega algoritma je, da zaporedoma pogleda vsak indeks v besedilu T in preveri, ali se vzorec P začne pri tem indeksu. Ustavimo se lahko že pri indeksu n – m. Namreč, če pričnemo preverjanje ujemanja pri indeksu večjem od n – m, je preostanek besedila T prekratek, da bi se še lahko ujemal z vzorcem niza P.
Če idejo napišemo shematsko, bo to:

za indekse od 0 do n - m
    preveri, ali se podniz dolžine m niza T z začetkom pri tem indeksu ujema z vzorcem P.

Preverjanje ujemanja podniza bomo izvedli tako, da bomo zaporedoma primerjali znak po znak. Če bomo naleteli na znaka, ki se ne ujemata, potem se podniz s tem začetkom ne ujema z vzorcem in to preverjanje takoj končamo. Če pa bomo s primerjanjem prišli do konca vzorca P, smo našli podniz, ki se z vzorcem P ujema. Če pri nobenem preverjanju ne bomo prišli do konca, potem vzorca v besedilu ni in vrnemo -1.

1.1.1 Primeri

Primer 1
Denimo, da imamo dano besedilo T = "monotonost". Iščemo, ali besedilo T vsebuje tako podzaporedje znakov, da se znak po znaku ujema z vzorcem P = "onos". Oglejmo si, kako izvedemo tako iskanje s pomočjo algoritma Groba Sila.

T	m	o	n	o	t	o	n	o	s	t
P	o	n	o	s						
j	0	1	2	3						
i	0	1	2	3	4	5	6	7	8	9

Algoritem Groba Sila išče ujemanje vzorca P v besedilu T tako, da preverja vsak znak posebej.

Korak 0:
Na prvem koraku vzamemo prvi znak iz besedila T, znak 'm', in ga primerjamo s prvim znakom iz vzorca P, znakom 'o'. Ker znaka nista enaka, celoten vzorec P premaknemo za en znak v desno.

T	m	o	n	o	t	o	n	o	s	t
P		o	n	o	s					
j		0	1	2	3					
i	0	1	2	3	4	5	6	7	8	9

Korak 1a do 1d:
Naslednja dva znaka, ki jih primerjamo, sta znaka na mestu T[1], znak 'o' in P[0], znak 'o'. Znaka sta enaka. Povečamo indeksa i in j za ena. Na koraku 1b primerjamo, ali je znak 'n' na mestu T[2] enak znaku 'n' na mestu P[1]. Znaka sta enaka. Povečamo indeksa i in j za ena in primerjamo znak na mestu T[3] ('o') z znakom na mestu P[2] ('o'). Tudi ta dva znaka sta enaka. Spet povečamo indeksa i in j za ena. Pogledamo, ali je znak na mestu T[4], znak 't', enak znaku na mestu P[3], znaku 's'. Znaka nista enaka. Celoten vzorec P premaknemo za en znak v desno tako, da v naslednjem koraku primerjamo znak na mestu T[2] z znakom na mestu P[0].

T	m	o	n	o	t	o	n	o	s	t
P			o	n	o	s				
j			0	1	2	3				
i	0	1	2	3	4	5	6	7	8	9

Korak 2:
Primerjamo znaka na mestu T[2] ('n') in P[0] ('o'). Znaka nista enaka. S celotnim vzorcem P se premaknemo za en znak v desno.

T	m	o	n	o	t	o	n	o	s	t
P				o	n	o	s			
j				0	1	2	3			
i	0	1	2	3	4	5	6	7	8	9

Korak 3a, 3b:
Primerjamo znaka na mestu T[3] ('o') in P[0] ('o'). Znaka sta enaka. Povečamo indeksa i in j za ena in primerjamo znak na mestu T[4] ('t') z znakom na mestu P[1] ('n'). Znaka nista enaka. S celotnim vzorcem P se premaknemo za en znak v desno.

T	m	o	n	o	t	o	n	o	s	t
P					o	n	o	s		
j					0	1	2	3		
i	0	1	2	3	4	5	6	7	8	9

Korak 4:
Primerjamo znaka na mestu T[4] ('t') in P[0] ('o'). Znaka nista enaka. S celotnim vzorcem P se premaknemo za en znak v desno.

T	m	o	n	o	t	o	n	o	s	t
P						o	n	o	s	
j						0	1	2	3	
i	0	1	2	3	4	5	6	7	8	9

Korak 5a do 5d:
Primerjamo znaka na mestu T[5] ('o') in P[0] ('o'). Znaka sta enaka. Povečamo indeksa i in j za ena in primerjamo znak na mestu T[6] ('n') z znakom na mestu P[1] ('n'). Znaka sta enaka. Povečamo indeksa i in j za ena in primerjamo znak na mestu T[7] ('o') z znakom na mestu P[2] ('o'). Znaka sta enaka. Povečamo indeksa i in j za ena in primerjamo znak na mestu T[8] ('s') z znakom na mestu P[3] ('s'). Znaka sta enaka. Ker smo prišli do konca vzorca P, ugotovimo, da smo v besedilu T našli tak podniz, ki se znak po znaku ujema z vzorcem P. Algoritem vrne celo število 5, ki označuje začetno mesto ujemanja vzorca P v besedilu T.

Primer 2

Podano imamo besedilo T = "algoritem za iskanje". Iščemo, ali je vzorec P = "iskan" podniz danega besedila T.
Dolžina besedila T je n = 20. Dolžina iskanega niza P je m = 5.
Najprej zapišimo niza v tabelo.

T	a	l	g	o	r	i	t	e	m		z	a		i	s	k	a	n	j	e
P	i	s	k	a	n															

Korak 0:
Vzamemo prvi znak v besedilu T, to je znak 'a' in pogledamo, ali je enak prvemu znaku v vzorcu P. Prvi znak v vzorcu P je znak 'i'. Znaka nista enaka. Celoten vzorec P premaknemo za 1 mesto naprej.

T	a	l	g	o	r	i	t	e	m		z	a		i	s	k	a	n	j	e
P		i	s	k	a	n														

Korak 1:
Vzamemo naslednji znak v besedilu T, znak 'l' in pogledamo, ali se znak ujema s prvim znakov v vzorcu P, znakom 'i'. Znaka nista enaka. Celoten vzorec P premaknemo za 1 mesto naprej.

T	a	l	g	o	r	i	t	e	m		z	a		i	s	k	a	n	j	e
P			i	s	k	a	n													

Korak 2, Korak 3, Korak 4:
Postopek je enak kot pri koraku 1 in koraku 2. Ko opravimo Korak 5, pridemo do naslednjega stanja.

T	a	l	g	o	r	i	t	e	m		z	a		i	s	k	a	n	j	e
P						i	s	k	a	n										

Korak 5:
Na tem koraku opazimo, da je znak 'i' v besedilu T enak znaku 'i' v vzorcu P. Zato vzamemo naslednji znak v besedilu T, znak 't', in ga primerjamo z naslednjim znakom v vzorcu P, znakom 's'. Opazimo, da ujemanja ni. Zato nadaljujemo na naslednjem koraku in se s prvim znakom vzorca P postavimo na mesto, kjer v besedilu T stoji znak 't'.

T	a	l	g	o	r	i	t	e	m		z	a		i	s	k	a	n	j	e
P							i	s	k	a	n									

Postopek nadaljujemo. Oglejmo si, kaj se zgodi na koraku 13. Stanje, preden izvedemo korak 13, je:

T	a	l	g	o	r	i	t	e	m		z	a		i	s	k	a	n	j	e
P														i	s	k	a	n		

Korak 13:
Znak v besedilu T je 'i'. Primerjamo ga s prvim znakom v vzorcu P, znakom 'i'. Našli smo ujemanje.
Korak 13a:
Znak v besedilu T je 's'. Enak je drugemu znaku v vzorcu P, znaku 's'.
Korak 13b, Korak 13c, Korak 13d:
Na vseh korakih se znak v T-ju ujema z znakom v P-ju.

Oglejmo si podrobneje **Korak 13d**:
Primerjamo znak v besedilu T z zadnjim znakom v vzorcu P. Znaka sta enaka. Ker vzorec P ne vsebuje več znakov, smo primerjanja končali. Ugotovili smo, da v besedilu T obstaja podniz, ki se znak po znaku ujema z vzorcem P. V tem primeru bi algoritem vrnil začetni indeks, pri katerem smo celotno primerjanje vzorca začeli, torej 13.

Primer 3
Denimo, da v besedilu T zamenjamo znak z indeksom 13 ('i') z znakom 'x'.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
T[i]	a	l	g	o	r	i	t	m	i		z	a		x	s	k	a	n	j	e

j	0	1	2	3	4
P[j]	i	s	k	a	n

Prvih 13 korakov je enakih, pri koraku 13 pa pridemo do stanja

T		x	s	k	a	n	j	e
P		i	s	k	a	n		

To bi za Korak 13 pomenilo, da se znaka ne ujemata in z vzorcem P se premaknemo za eno mesto naprej.

T		x	s	k	a	n	j	e
P			i	s	k	a	n	

Tudi tu takoj ni ujemanja in gremo na korak 14.

T		x	s	k	a	n	j	e
P				i	s	k	a	n

Ker ujemanja ni, se premaknemo za korak naprej.

T		x	s	k	a	n	j	e	
P					i	s	k	a	n

Iz tabele je razvidno, da je dolžina vzorca P večja kot dolžina preostanka besedila T. Ugotovili smo, da v besedilu T ne obstaja tak podniz, ki bi se v celoti, znak po znaku, ujemal z iskanim vzorcem P. V tem primeru bi algoritem vrnil vrednost -1.

1.1.2 Algoritem

Algoritem je sestavljen iz dveh gnezdenih zank. S prvo, zanko `for`, se sprehodimo po dolžini besedila T. Tekla bo od indeksa `0` do `(n - m)`. V spremenljivki `n` tipa `int` hranimo dolžino besedila T, v `m`, ki bo tudi tipa `int`, pa dolžino vzorca P. Indeks `(n - m)` označuje zadnje mesto, kjer je še smiselno preverjati ujemanje znakov. Če se na tem mestu znaka ne ujemata, pomeni, da je vzorec P večji od preostanka besedila T in lahko končamo z odgovorom `-1` (vzorca torej ni v besedilu). Ta zanka torej preverja, če se podniz T[i .. i + m - 1] ujema z vzorcem P. To preverimo z notranjo zanko. Tu se s pomočjo spremenljivke `j` sprehodimo po dolžini vzorca P. Dokler je še kaj znakov v vzorcu P, torej velja `j < m`, in je znak v T enak znaku v P, bomo povečevali indeks `j`. Po končanem primerjanju bomo pogledali vrednost v indeksu `j`. Če je vrednost enaka dolžini niza P, torej `m`, pomeni, da smo v besedilu T našli podniz, ki se znak po znaku ujema z iskanim vzorcem P. V tem primeru pogledamo vrednost v spremenljivki `i`, ki označuje začetno mesto ujemanja v besedilu T in jo vrnemo. S tem zaključimo metodo.
Če se je zunanja zanka `for` iztekla, vrednost v spremenljivki `j` ni bila nikoli enaka vrednosti v spremenljivki `m`, saj bi drugače metodo že zaključili. To pomeni, da ujemanja nismo našli in vrnemo `-1`.

```
import java.util.*;
public class TekstovniAlgoritmi
{
 public static int algoritemGrobaSila(String t, String p)
  {
    int n = t.length(); //dolžina besedila T
    int m = p.length(); //dolžina vzorca P

    for(int i = 0; i <= (n-m); i++) //sprehodimo se po besedilu T
    {
      int j = 0; //sprehajanje po vzorcu P
      while(j < m && t.charAt(i+j) == p.charAt(j))
      {
        j = j + 1; //znaka se ujemata, primerjamo naslednja
      }
      if(j == m) // vsi znaki v P so se ujemali s tistimi v T
        return i; //našli smo ujemanje na i-tem mestu
    }
    return -1; //nismo našli ujemanja, vrnemo -1
  }
}
```

1.1.3 Časovna zahtevnost

Oglejmo si časovno zahtevnost algoritma. Osnovna operacija pri iskanju podniza je primerjanje posameznih znakov. Algoritmi se bodo med seboj razlikovali glede na to, koliko teh primerjanj opravijo.
Velikost problema, ki ga obravnavamo, je odvisna od dolžine obeh nizov, ki sta vhodna parametra. Kot vhodna parametra algoritem sprejme besedilo T in niz P. Dolžina T, ki jo označimo z `n` in P, ki jo označimo z `m`, predstavljata velikost problema. Za algoritem groba sila, ki ga označimo z `a`, določimo časovno zahtevnost problema $T_a(n, m)$.

Algoritem je sestavljen iz zanke `for` in gnezdene zanke `while`. Zunanja zanka se sprehaja po vseh možnih začetnih indeksih vzorca P v besedilu T. Notranja zanka se sprehaja po indeksih vzorca P in primerja, ali se znak v vzorcu ujema z danim nizom. Največ naredimo $(n - m + 1) * m$ primerjanj, najmanj pa m, če se vzorec P nahaja že na začetku besedila T.
Oglejmo si, kakšna je časovna zahtevnost algoritma v najboljšem, povprečnem in najslabšem primeru.

Najboljši primer
Če zanemarimo očitno nesmiselen primer, ko je dolžina vzorca P daljša od dolžine besedila T, se v najboljšem primeru vzorec P nahaja na začetku besedila T. Takrat dolžina besedila T ne vpliva na časovno zahtevnost. Za vsak znak v vzorcu P opravimo eno primerjavo. Torej je število primerjanj enako m.

$$T_a(n, m) = O(m)$$

Najslabši primer
V najslabšem primeru vzorca v besedilu sploh ni. Pri tem bo najslabši primer nastopil takrat, ko bomo pri preverjanju v notranji podzanki vedno prišli prav do konca. Premislek nam pokaže, da bo tak najslabši par denimo "xxxxxxxxz" in "xxy" ali "xxxxxxxx" in "xxy". Torej mora biti besedilo T takšno, da se $m - 1$ zaporednih znakov iz besedila T ujema z $m - 1$ znakov iz vzorca P, m-ti znak iz besedila P pa se ne ujema z znakom iz T. V tem primeru se izvede maksimalno število primerjanj.

$$T_a(n, m) = O((n - m + 1) * m) = O(n * m)$$

Pričakovana časovna zahtevnost

Analiza pričakovane časovne zahtevnosti je nekoliko zapletena, saj bi morali oceniti verjetnost, da se bo vzorec znašel v besedilu, upoštevati sestavo vzorcev in besedila ... Zato povejmo le, da je tudi enaka `O(n * m)`.

Kadar je dolžina niza P sorazmerna polovici dolžine besedila T, torej `m = n/2`, je metoda kvadratične časovne zahtevnosti

$$T_a(n, m) = O(n * m) = O(n * (n/2)) = O((n * n)/2) = O(n^2)$$

1.2 Algoritem Boyer-Moore

Algoritem Boyer-Moore za iskanje podniza v danem nizu uporablja drugačen pristop kot algoritem Groba Sila. Pri preverjanju, ali se določen podniz besedila T ujema z vzorcem, začne preverjanje pri zadnji črki podniza P in se premika proti začetku vzorca P. To je tako imenovana tehnika zrcalo (Looking-Glass).
Druga tehnika, ki je uporabljena v algoritmu, je tehnika preskakovanja znakov (Character-Jump). Če algoritem ugotovi, da se ustrezna znaka v vozorcu in besedilu ne ujemata, vzorec P premakne za k znakov v desno. Pri tem k določimo glede na določene lastnosti vzorca, besedila in primerjanega znaka.

Oglejmo si, zakaj je preverjanje od zadaj dober pristop.
Vzemimo besedilo T = "V gorah je sneg.". Iščemo, ali je vzorec P = "gorah" vsebovan v besedilu T.
Zapišimo niza v tabelo.

T	V		g	o	r	a	h		j	e		s	n	e	g	.
P	g	o	r	a	h											

Če uporabimo algoritem Groba Sila, začnemo preverjanja v prvem znaku iz vzorca P in ga primerjamo s prvim znakom iz besedila T, torej primerjamo 'g' z 'V'. Ker znaka nista enaka, se z vzorcem P premaknemo za en znak v desno.

T	V		g	o	r	a	h		j	e		s	n	e	g	.
P		g	o	r	a	h										

Primerjamo znaka ' ' in 'g'. Znaka nista enaka, zato se premaknemo z vzorcem P za en znak v desno.

T	V		g	o	r	a	h		j	e		s	n	e	g	.
P			g	o	r	a	h									

Primerjamo znaka 'g' in 'g'. Ker sta enaka primerjamo naslednja znaka iz vzorca P in besedila T. Postopek ponavljamo, dokler ne pridemo do konca vzorca P.

Oglejmo si pristop od zadaj.
Sedaj začnemo s primerjanji v zadnjem znaku vzorca P in ga primerjamo z znakom na mestu m v besedilu T, torej primerjamo 'h' z 'r'.

T	V		g	o	r	a	h		j	e		s	n	e	g	.
P	g	o	r	a	h											

Znaka nista enaka. Ker znak 'r' nastopa v vzorcu P, se s celotnim vzorcem P lahko premaknemo v desno tako, da poravnamo znak 'r' iz vzorca P z znakom 'r' iz besedila T. Ustrezne premike za vse znake v nizu P določimo vnaprej.

T	V		g	o	r	a	h		j	e		s	n	e	g	.
P			g	o	r	a	h									

Vidimo, da smo v tem primeru s pomočjo pristopa od zadaj hitreje prišli do rešitve. Izkaže se, da je to res v večini primerov.

Dejansko bi lahko nekaj takega naredili tudi, kadar znake primerjamo od spredaj. Primerjali bi znak na mestu P[0] z znakom na mestu T[0], torej 'g' z 'V'. Znaka nista enaka. Pogledamo ali besedilo T vsebuje znak 'g'. Ker ga vsebuje, se z vzorcem P premaknemo tako, da 'g' poravnamo z 'g'.
Vendar to ni pameten pristop. V tem primeru bi obdelovali besedilo T. To pomeni, da bi morali iskati pojavitve znakov v besedilu T. Ker pa je dolžina besdila T običajno bistveno daljša od dolžine vzorca P, bi bil tak pristop časovno zahtevnejši.

Oglejmo si nekaj primerov premikov, ki jih naredimo s pomočjo tehnike preskakovanja znakov. Zakaj so taki in kako jih določimo, si bomo ogledali v nadaljevanju.

Zgled 1:
Denimo, da imamo podano besedilo T = "abcdab" in vzorec P = "cda".

T	a	b	c	d	a	b
P	c	d	a			

Algoritem Boyer-Moore uporablja tehniko zrcalo, kar pomeni, da preverjanja začnemo v zadnji črki vzorca P, to je 'a'. Znak primerjamo z znakom 'c' iz besedila T. Znaka se ne ujemata. Sedaj nastopi tehnika preskakovanja znakov. Pogledamo, ali vzorec P vsebuje znak 'c'. Ker ga vsebuje, premaknemo vzorec P tako, da se znak 'c' iz vzorca P poravna z znakom 'c' iz besedila T.

T	a	b	c	d	a	b
P			c	d	a	

Zgled 2:
Denimo, da imamo podano besedilo T = "abcdab" in vzorec P = "kda".

T	a	b	c	d	a	b
P	k	d	a			

Primerjanja začnemo z zadnjim znakom v vzorcu P, znakom 'a' in ga primerjamo z znakom 'c' iz besedila T. Znaka se ne ujemata. Pogledamo, ali vzorec P vsebuje znak 'c'. Ker ga ne vsebuje, se s celotnim vzorcem P premaknemo le za eno mesto v desno.

T	a	b	c	d	a	b
P				k	d	a

Zgled 3:
Denimo, da imamo podano besedilo T = "jablana" in vzorec P = "ana". Po nekaj korakih pridemo do naslednjega stanja:

T	j	a	b	l	a	n	a
P				a	n	a	

Primerjanje začnemo z zadnjim znakom v vzorcu P, znakom 'a'. Primerjamo ga z znakom 'n' iz besedila T. Znaka nista enaka. Sedaj preverimo, ali vzorec P vsebuje znak 'n'. Ker ga vsebuje, se s celotnim vzorcem P premaknemo tako, da znak 'n' iz vzorca P poravnamo z znakom 'n' iz besedila T.

T	j	a	b	l	a	n	a
P					a	n	a

Zgled 4:
V danem besedilu T = "tipkovnica in miška", iščemo vzorec P = "klovn". Po nekaj korakih pridemo do naslednjega stanja:

T	t	i	p	k	o	v	n	i	c	a		i	n		m	i	š	k	a
P			k	l	o	v	n												

Vzamemo zadnji znak iz vzorca P, znak 'n' in ga primerjamo z znakom 'n' iz besedila T. Znaka sta enaka. Primerjamo naslednji znak iz vzorca P, znak 'v' z znakom 'v' iz besedila T. Ker sta znaka enaka, primerjamo naslednja znaka, znaka 'o'. Ponovno smo našli ujemanje, zato primerjamo znaka 'l' in 'k'. Znaka nista enaka. Ker vzorec P vsebuje znak 'k', se z vzorcem P premaknemo tako, da poravnamo znak 'k' iz vzorca P z znakom 'k' iz besedila T.

T	t	i	p	k	o	v	n	i	c	a		i	n		m	i	š	k	a
P				k	l	o	v	n											

Zgled 5:
Popravimo iskani vzorec P iz zgornjega zgleda. Naj bo sedaj P = "plovn". Še vedno iščemo ali vzorec P obstaja v besedilu T = "tipkovnica in miška". Po nekaj korakih pridemo do naslednjega stanja:

T	t	i	p	k	o	v	n	i	c	a		i	n		m	i	š	k	a
P			p	l	o	v	n												

Znake primerjamo na enak način kot na zgornjem zgledu. Podrobneje si oglejmo korak na katerem primerjamo znak 'l' iz vzorca P z znakom 'k' iz besedila T. Znaka nista enaka. Ker vzorec P ne vsebuje znaka 'k', se s celotnim vzorcem P premaknemo za en znak v desno od mesta neujemanja.

T	t	i	p	k	o	v	n	i	c	a		i	n		m	i	š	k	a
P					p	l	o	v	n										

Ustrezne premike smo seveda "videli". Sedaj pa si oglejmo, kako določimo premike, ki smo jih prikazali na zgornjih zgledih. Te premike bomo potem lahko uporabili v algoritmu.

Vrednost, ki pove, za koliko znakov premaknemo vzorec P v desno, označimo s spremenljivko k. Premik za k znakov v desno pove, kako daleč naj premaknemo vzorec P, da se znak P[j] ujema z znakom T[i]. Pri tem je j indeks, ki pove, na katerem mestu v vzorcu P trenutno primerjamo znak z znakom iz besedila T. V besedilu T smo na mestu it (primerjamo torej T[it] in P[j]). Mesto i (v besedilu T), kjer se znak T[i] ujema z znakom P[j], izračunamo po formuli

```
i = it + m - min(j, 1 + L(T[it]))                    [BM1],
```

ki jo označimo z BM1. Tu je L(x) funkcija, ki jo bomo še opisali.
Novi indeks i ni nikoli manjši od trenutnega indeksa it. To pomeni, da premik izvedemo vedno v desno od trenutnega indeksa it. Največji premik opravimo takrat, ko celotni vzorec P ne vsebuje znaka T[it]. Vrednost funkcije L za ta znak je namreč -1. Tedaj se s celotnim vzorcem P premaknemo za eno v desno od znaka T[it]. Če je torej to že takoj na začetku primerjanja (od zadaj), se torej premaknemo za celotno dolžino vzorca P.
Za implementacijo tehnike preskakovanja znakov potrebujemo posebno funkcijo, ki jo poimenujemo L(x). Ta vzame znak x iz dane abecede in pogleda, ali vzorec P vsebuje znak x. Če ga vsebuje, vrne zadnje mesto, na katerem najdemo znak x v P, sicer vrne -1.

Funkcija L(x)

Dano imamo besedilo T in vzorec P. Abeceda A, ki jo bomo uporabili pri funkciji, je sestavljena iz tistih znakov, ki nastopajo v besedilu T. Pri gradnji funkcije L(x) velja:

- x je eden izmed znakov abecede A.
- L(x) označuje tisto mesto (indeks), kjer v iskanem vzorcu P opazimo x. Če se znak x v vzorcu P pojavi večkrat, za L(x) vzamemo najbolj desno mesto (največji indeks) .
  Primer: Vzorec P = "ogledalo". Naj bo x = 'l'. Znak 'l' se pojavi v vzorcu P dvakrat, in sicer na mestu z indeksom 2 in 6. Za L('l') vzamemo najbolj desno mesto, torej 6.
- Če se x v P-ju ne pojavi, funkcija L(x) vrne -1.
  Primer: Vzorec P = "jabolko". Naj abeceda A vsebuje znake {a, b, e, j, k, l, o}. Vzemimo znak 'e' iz abecede A. Vzorec P ne vsebuje znaka 'e'. To pomeni, da je L('e') enak -1.

Poglejmo si konstrukcijo te funkcije na nekaj zgledih.

Zgled 1

Dano imamo besedilo T = "algoritem za iskanje" in vzorec P = "iskanje". Abeceda je sestavljena iz znakov iz besedila T, A = {a, e, g, i, j, k, l, m, n, o, r, t}.

Zapišimo vzorec P v tabelo.

j	0	1	2	3	4	5	6
P	i	s	k	a	n	j	e

Za vsak znak iz abecede A bomo določili, če vzorec P vsebuje ta znak. Če ga, v tabelo vpišemo indeks, kjer se znak pojavi, sicer pa vpišemo -1.
Vzemimo prvi znak iz abecede A, znak 'a'. P vsebuje znak 'a' na mestu z indeksom 3. Ugotovitev zapišemo v tabelo.

A	a	e	g	i	j	k	l	m	n	o	r	t
L(x)	3											

Naredimo to za vse znake abecede A. Pridemo do naslednjega rezultata:

A	a	e	g	i	j	k	l	m	n	o	r	t
L(x)	3	6	-1	0	5	2	-1	-1	4	-1	-1	-1

Zgled 2

Denimo, da je T = "Sneg je pobelil skoraj celo Slovenijo." in vzorec P = "pobelil". Pri abecedi A moramo paziti na to, da ločimo velike in male črke, velja torej 'S' ≠ 's'. Pika je tudi znak, ki spada k abecedi A. Abeceda A = {a, b, c, e, g, i, j, k, l, n, o, p, r, s, S, .}.
Zapišimo vzorec P v tabelo.

j	0	1	2	3	4	5	6
P	p	o	b	e	l	i	l

Ko računamo L('l') opazimo, da se v vzorcu P znak 'l' ponovi dvakrat, in sicer na mestu z indeksom 4 in na mestu z indeksom 6. V tem primeru moramo upoštevati znak 'l', ki je najbolj desno, torej znak 'l' z indeksom 6. L('l') je torej 6.

Funkcija L(x) je enaka:

A	a	b	c	e	g	i	j	k	l	n	o	p	r	s	S	.
L(x)	-1	2	-1	3	-1	5	-1	-1	6	-1	1	0	-1	-1	-1	-1

Preskakovanje znakov

Pri preskakovanju znakov pogosto naletimo na dve možnosti. Čeprav bomo v samem algoritmu neposredno uporabili kar formulo BM1, pa se bomo v razlagi zgledov večkrat sklicali kar na ti dve možnosti. Označili ju bomo s P1 in P2.

P1:

Pogledamo, ali P vsebuje znak T[i]. Če P vsebuje ta znak, denimo na k-tem mestu, se s P-jem premaknemo tako, da se znak na mestu k, torej P[k], poravna z znakom na mestu i (torej z znakom T[i]).
Vemo, da vzorec P vsebuje znak T[i] na mestu k. Če vzorec P premaknemo tako, da sta znaka T[i] in P[k] poravnana, je možno, da bomo na naslednjem koraku našli ujemanje.
Oglejmo si primer takega premika.
Dano imamo besedilo T = "algoritem". Iščemo vzorec P = "gor". Zapišimo niza v tabelo.

T	a	l	g	o	r	i	t	e	m
P	g	o	r						

Iskanje začnemo v zadnjem znaku vzorca P, znaku 'r' in pogledamo, ali se znak 'r' ujema z znakom 'g'. Ker se znaka ne ujemata, opravimo preskok. Zato preverimo, ali v vzorcu P obstaja znak 'g'. Vidimo, da znak obstaja in je na mestu P[0]. Opraviti imamo torej z možnostjo **P1**. P premaknemo za 2 znaka v desno tako, da se P[0] ujema s T[2]. Če vzorec P premaknemo tako, da sta znaka na mestu T[2] in P[0] poravnana, obstaja možnost, da bomo na naslednjem koraku našli ujemanje. To se v našem zgledu res zgodi

T	a	l	g	o	r	i	t	e	m
P			g	o	r				

Oglejmo si podrobneje ta premik tipa P1. Rekli smo, da za premik uporabimo formulo [BM1]. Poglejmo, kaj nam glede premika pove ta formula. Za znak 'g' je L('g') = 0. Trenutni indeks it mesta v besedilu T, kjer preverjamo ujemanje z znakom iz P, je 2. Tudi indeks v P (torej j) je 2. Dolžina vzorca P (torej m) je 3. Formula [BM1] nam pove indeks, na katerem bo tisti znak v T-ju, ki ga bomo v naslednjem koraku primerjali z znakom P[2].

```
i = it + m - min(j, 1 + L(T[it])) = 2 + 3 - min(2, 1 + 0) = 4.
```

Iz tega sledi, da moramo P premakniti tako, da bomo P[2] primerjali z znakom T[4].

Kaj pa, če P vsebuje znak na mestu T[i] na h-tem in k-tem mestu (in je h < k)? V tem primeru poravnamo vzorec P z besedilom T tako, da vzamemo najbolj desni znak iz vzorca P, torej znak na mestu k in ga poravnamo z znakom na mestu T[i].
Oglejmo si primer takega premika.
Dano imamo besedilo T = "krema". Iščemo, ali besedilo T vsebuje vzorec P = "mama".

T	k	r	e	m	a
P	m	a	m	a	

Ker znaka 'm' in 'a' nista enaka, pogledamo, ali vzorec P vsebuje znak 'm'. P vsebuje znak 'm' na ničtem in drugem mestu. Vzorec P moramo premakniti tako, da se znak 'm' iz besedila T ujema z najbolj desnim znakom 'm' iz vzorca P. To je znak 'm' z indeksom 2.

T	k	r	e	m	a
P		m	a	m	a

Poglejmo še, kaj nam glede tega premika pove formula [BM1]. Tako je za znak 'm' L('m') = 2. Trenutni indeks it mesta v besedilu T, kjer preverjamo ujemanje z znakom iz P, je 3. Tudi indeks v P (torej j) je 3. Dolžina vzorca P (torej m) je 4. Formula [BM1] nam pove indeks, na katerem bo tisti znak v T-ju, ki ga bomo v naslednjem koraku primerjali z znakom P[3].

```
i = it + m - min(j, 1 + L(T[it])) = 3 + 4 - min(3, 1 + 2) = 4.
```

Iz tega sledi, da moramo P premakniti tako, da bomo P[3] primerjali z znakom T[4].

Poglejmo zakaj moramo uporabiti najbolj desni znak 'm' iz vzorca P.
Denimo, da imamo besedilo T = "kmama" in vzorec P = "mama".

T	k	m	a	m	a
P	m	a	m	a	

Znak 'a' iz besedila T ni enak znaku 'm' iz vzorca P. P vsebuje znak 'm' na mestu P[0] in P[2].

Poglejmo kaj se zgodi, če poravnamo P s T tako, da vzamemo najbolj levi 'm' iz P. Torej znak 'm' na mestu P[0].

T	k	m	a	m	a		
P				m	a	m	a

Ker je vzorec P daljši od preostanka besedila T, bi algoritem vrnil vrednost -1. Torej ujemanja nismo našli. To pa ni res. Že od daleč vidimo, da ujemanje obstaja.

Sedaj pa vzemimo najbolj desni znak 'm' iz vzorca P, znak na mestu P[3].

T	k	m	a	m	a
P		m	a	m	a

Našli smo ujemanje.

Iz primera sledi, da moramo vedno vzeti najbolj desni znak, sicer je možno, da izpustimo rešitev.

Možnost P2:
Če **P1** ne nastopi, P ne vsebuje znaka T[it]. Tedaj premaknemo celoten vzorec P tako, da se prvi znak iz vzorca P poravna z znakom T[it + 1].
Denimo, da je vzorec P dolžine m. Vemo, da primerjanja začnemo pri zadnjem znaku v vzorcu P, P[m - 1] in ga primerjamo z znakom na mestu T[m - 1]. Če znaka nista enaka in vzorec P ne vsebuje znaka T[m - 1] vemo, da ujemanja zagotovo ne bomo našli od mesta 0 do mesta m - 1 v besedilu T. Zato lahko celoten vzorec P premaknemo tako, da se prvi znak iz vzorca P poravna z znakom na mestu T[m].
Oglejmo si primer takšnega premika.
Dano imamo besedilo T = "najpogostejše". Iščemo vzorec P = "gost". Zapišimo niza v tabelo.

T	n	a	j	p	o	g	o	s	t	e	j	š	e
P	g	o	s	t									

Znak 'p' iz besedila T in znak 't' iz vzorca P se ne ujemata. Pogledamo, ali vzorec P vsebuje znak 'p'. Ker ga ne vsebuje, je za premik nastopila možnost P2. To pomeni, da celoten vzorec P premaknemo tako, da je znak na mestu P[0] poravnan z znakom na mestu T[3]. Tak premik lahko naredimo zato, ker vidimo, da zagotovo ne najdemo ujemanja od mesta T[0] do T[3], saj vsaj en znak iz besedila T ne obstaja v vzorcu P.

T	n	a	j	p	o	g	o	s	t	e	j	š	e
P					g	o	s	t					

Poglejmo, kaj nam glede premika pove formula [BM1]. Tako je za znak 'p' L('p') = -1. Trenutni indeks it mesta v besedilu T, kjer preverjamo ujemanje z znakom iz P, je 3. Tudi indeks v P (torej j) je 3. Dolžina vzorca P (m) je 4. Formula [BM1] nam pove indeks, na katerem bo tisti znak v T-ju, ki ga bomo v naslednjem koraku primerjali z znakom P[3]

```
i = it + m - min(j, 1 + L(T[it])) = 3 + 4 - min(3, 0) = 7.
```

Iz tega sledi, da moramo P premakniti tako, da bomo P[3] primerjali z znakom T[7], kar se ujema s prej napovedanim.

Oglejmo si še en zgled. Denimo, da v besedilu T = "*vremenoslovec" iščemo vzorec P = "okno".

T	*	v	r	e	m	e	n	o	s	l	o	v	e	c
P	o	k	n	o										

Po nekaj korakih pridemo do stanja

T	*	v	r	e	m	e	n	o	s	l	o	v	e	c
P					o	k	n	o						

Najprej pogledamo, ali sta znaka 'o' iz besedila T in 'o' iz vzorca P enaka. Ker sta znaka enaka, pogledamo naslednja dva znaka, znak 'n' iz besedila T in znak 'n' iz vzorca P. Tudi ta sta enaka. Naslednja dva znaka 'e' in 'k' pa nista enaka. Zato pogledamo, ali vzorec P vsebuje znak 'e'. Ker ga ne vsebuje, se z vzorcem P premaknemo tako, da prvi znak iz vzorca P poravnamo z znakom 'n' iz besedila T. Tudi tu se premaknemo za ena v desno od mesta neujemanja.

T	*	v	r	e	m	e	n	o	s	l	o	v	e	c
P							o	k	n	o				

Tudi tu si poglejmo, kaj nam glede premika pove formula [BM1]. Tako je za znak 'e' L(e) = -1. Trenutni indeks it mesta v besedilu T, kjer preverjamo, ujemanje z znakom iz P, je 5. Indeks v P (torej j) je 1. Dolžina vzorca P (m) je 4. Formula [BM1] nam pove indeks, na katerem bo tisti znak v T-ju, ki ga bomo v naslednjem koraku primerjali z znakom P[3].

```
i = it + m - min(j, 1 + L(x)) = 5 + 4 - min(1, 0) = 9.
```

Iz tega sledi, da moramo P premakniti tako, da bomo P[3] primerjali z znakom T[9], kar seveda spet ustreza naši napovedi.

Premislimo zakaj se premaknemo tako.
Z rdečim pravokotnikom označimo mesto neujemanja.

T	*	v	r	e	m	e	n	o	s	l	o	v	e	c
P					o	k	n	o						

Poglejmo kako bi lahko premaknili vzorec P, ne da bi izpustili morebitno ujemanje.
Vzorec P bi lahko premaknili tako, da poravnamo znak 'o' iz P z znakom 'e' iz T. Na ta način zagotovo ne bi izpustili morebitnega ujemanja vzorca P v besedilu T.

T	*	v	r	e	m	e	n	o	s	l	o	v	e	c
P						o	k	n	o					

Poglejmo si, če obstaja boljši primer premika. Vzorec P bi lahko premaknili tako, da poravnamo znak 'o' iz P z znakom 'n' iz T. Tako se premaknemo s pomočjo algoritma Boyer-Moore.

T	*	v	r	e	m	e	n	o	s	l	o	v	e	c
P							o	k	n	o				

Sedaj premislimo zakaj je drugi primer premika boljši od prvega.
Če bi upoštevali prvi premik, bi naredili en korak več kot je potrebno. Znak 'o' ni enak znaku 'e'. To pomeni, da na naslednjem koraku še ne bi našli ujemanja. Zato lahko celoten vzorec P premaknemo za eno mesto v desno od mesta neujemanja, torej upoštevamo drugi premik.

1.2.1 Primeri

Sedaj pridobljeno znanje preizkusimo na celotnih primerih iskanja:

Primer 1
Dano imamo besedilo T = "alge in gorenje". Iščemo, ali besedilo T vsebuje podniz"gor". Dolžina besedila T je 15, torej n = 15. Dolžina vzorca P, torej m, je 3.

Pripravimo si tabelo, v katero vpišemo vrednosti funkcije L(x). V tem primeru je abeceda A = {a, e, g, i, j, l, n, o, r, }. Zadnji znak v abecedi A je presledek. Spomnimo se, da L(x) gradimo tako, da pogledamo, ali vzorec P vsebuje znak iz abecede A. Če ga vsebuje, v tabelo vpišemo, na katerem mestu P vsebuje znak, sicer vpišemo -1.

x	a	e	g	i	j	l	n	o	r	
L(x)	-1	-1	0	-1	-1	-1	-1	1	2	-1

Besedilo T in vzorec P vpišimo v tabelo. Po vzorcu P se sprehajamo z indeksom j, po besedilu T pa z indeksom i.

T	a	l	g	e		i	n		g	o	r	e	n	j	e
P	g	o	r												
j	0	1	2												
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Korak 0:
Algoritem Boyer-Moore uporablja tehniko zrcalo. To pomeni, da preverjanja začne z zadnjim znakom iz vzorca P, torej znakom z indeksom m - 1 = 2. Znak na mestu P[j] = P[m - 1] = P[2], torej 'r', primerjamo z znakom na mestu T[i] = T[j] = T[2], torej z znakom 'g'. Velja P[2] ≠ T[2]. Pogledamo, ali vzorec P vsebuje znak 'g'. S pomočjo tabele L(x) ugotovimo, da vzorec P vsebuje znak 'g' na mestu z indeksom 0. Nastopi možnost **P1**, ki pravi, da celoten vzorec P premaknemo tako, da se znak na mestu P[0] poravna z znakom na mestu T[2].

T	a	l	g	e		i	n		g	o	r	e	n	j	e
P			g	o	r										
j			0	1	2										
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Korak 1:
Ponovno preverjanje začnemo z zadnjim znakom iz vzorca P, znakom 'r' in ga primerjamo z znakom z indeksom 4 iz besedila T, torej s ' ' (presledkom). Pogledamo, ali vzorec P vsebuje presledek. S pomočjo funkcije L(x) ugotovimo, da P presledka ne vsebuje. Nastopi **P2**. Celoten vzorec P premaknemo tako, da je znak na mestu P[0] poravnan z znakom na mestu T[5].

T	a	l	g	e		i	n		g	o	r	e	n	j	e
P						g	o	r							
j						0	1	2							
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Korak 2:
Zadnji znak iz vzorca P je enak znaku 'r', ki ga primerjamo z znakom ' ' v T na mestu 7. Znaka nista enaka. Funkcija L(x) nam pove, da vzorec P ne vsebuje presledka, zato nastopi **P2**. Celoten vzorec P premaknemo tako, da se znak na mestu P[0] poravna z znakom na mestu T[8].

T	a	l	g	e		i	n		g	o	r	e	n	j	e
P									g	o	r				
j									0	1	2				
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Korak 3a do Korak 3c:
Vzamemo zadnji znak iz vzorca P, znak 'r' in ga primerjamo z znakom 'r' iz besedila T. Znaka sta enaka. Indeksa i in j zmanjšamo za ena. Sledi korak 3b. Primerjamo znak z indeksom j = 1, torej znak 'o', z znakom z indeksom i = 9, z znakom 'o'. Znaka sta enaka. Sledi korak 3c. Zmanjšamo indeksa i in j za ena in primerjamo znak 'g' iz P z znakom 'g' iz T. Znaka sta enaka. Prišli smo do prvega znaka, znaka z indeksom 0 v P, torej se vzorec ujema z besedilom. Algoritem bi v tem primeru vrnil število 8, ki predstavlja začetni indeks ujemanja vzorca P v besedilu T.

Primer 2
T = "algoritem za iskanje", dolžina T n = 20
P = "iskanje", dolžina P m = 7

T	a	l	g	o	r	i	t	e	m		z	a		i	s	k	a	n	j	e
P	i	s	k	a	n	j	e													
j	0	1	2	3	4	5	6													
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Najprej si pripravimo tabelo, v katero vpišimo vrednosti funkcije L(x).

x	a	e	g	i	j	k	l	m	n	o	r	t
L(x)	3	6	-1	0	5	2	-1	-1	4	-1	-1	-1

Korak 0:
Primerjanje začnemo z indeksom m - 1 = 6. Znak P[j] = P[m - 1] = P[6] ('e') primerjamo z znakom na mestu i = m - 1= 6, torej znakom T[6] = 't'. Ker je T[6] ≠ P[6], pogledamo, ali P vsebuje znak 't'. S pomočjo tabele L(x) hitro ugotovimo, da ga ne vsebuje, saj je L('t') = -1. Torej je nastopil možnost P2 tehnike preskakovanja znakov. Ta pravi, da se premaknemo tako, da P[0] poravnamo s T[7]. Potrdimo, da res opravimo tak premik, še preko formule [BM1]. Indeks i bo torej it + m - min(j, 1 + L('t')) = 6 + 7 - min(6, 1 + (-1)) = 13 - min(6, 0) = 13 - 0 = 13. Iz tega sledi, da vzorec P premaknemo tako, da se znak na mestu P[6] ujema z znakom na mestu T[13], kar je seveda isto, kot če P[0] poravnamo s T[7].

T	a	l	g	o	r	i	t	e	m		z	a		i	s	k	a	n	j	e
P								i	s	k	a	n	j	e						
j								0	1	2	3	4	5	6						
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Korak 1:
Ponovno vzamemo zadnji znak v vzorcu P, znak 'e' in ga primerjamo z znakom z indeksom 13 v besedilu T, znakom 'i'. Znaka se ne ujemata. Pogledamo, ali P vsebuje znak 'i'. S pomočjo zgrajene tabele L(x) vidimo, da ga vsebuje. Torej je nastopila možnost P1. Znak se nahaja na mestu z indeksom 0. P premaknemo tako, da znak na mestu P[0] poravnamo z znakom na mestu T[13]. Poglejmo, kaj pravi formula [BM1]. Indeks i bo enak izračunu: it + m - min(j, 1 + L('i')) = 13 + 7 - min(6, 1 + (0)) = 13 + 7 - 1 = 19. To pomeni, da znak na mestu P[6] poravnamo z znakom na mest T[19]. To je enako temu da P[0] poravnamo s T[13].

T	a	l	g	o	r	i	t	e	m		z	a		i	s	k	a	n	j	e
P														i	s	k	a	n	j	e
j														0	1	2	3	4	5	6
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Korak 2a do Korak 2g:
Vzamemo zadnji znak v vzorcu P, znak 'e' in ga primerjamo z zadnjim znakom iz besedila T, znakom 'e'. Zanka sta enaka. Trenutno je i = 19, j = 6. Indeksa i in j zmanjšamo za ena. Sledi korak 2b. Primerjamo znak z indesom j - 1 = 5 v P z znakom z indeksom i - 1 = 18. Znaka sta enaka, zato zmanjšamo i in j za ena. Postopek ponavljamo, dokler ne pridemo do znaka z indeksom j = 0 v P. Takrat nastopi Korak 2g. Pogledamo, ali se znak na mestu P[0] ujema z znakom na mestu T[13]. Znaka sta enaka. Po vzorcu P se ne moremo več premikati v levo, saj smo prišli do prvega znaka, znaka z indeksom 0 v P. Algoritem bi v tem primeru vrnil število 13, ki predstavlja začetni indeks ujemanja vzorca P v besedilu T.

Primer 3
T = "Rdeča zora - mokra gora.", dolžina T n = 24.
P = "skupaj jemo", dolžina P m = 13.

Napišimo rezultate funkcije L(x) v tabelo.

x	a	č	d	e	k	m	o	r	z	R		-	.
L(x)	4	-1	-1	8	-1	9	10	-1	-1	-1	6	-1	-1

Niza zapišimo v tabelo.

T	R	d	e	č	a		z	o	r	a		-		m	o	k	r	a		g	o	r	a	.
P	s	k	u	p	a	j		j	e	m	o													
j	0	1	2	3	4	5	6	7	8	9	10													
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23

Korak 0:
Primerjanje začnemo z zadnjim znakom iz vzorca P, znakom 'o' in ga primerjamo z znakom ' ' iz besedila T. Znaka nista enaka. Funkcija L(x) nam pove, da vzorec P vsebuje znak presledek na mestu z indeksom 6, torej P[6]. Celoten vzorec P premaknemo tako, da se znak v P z indeksom 6 poravna z znakom v T z indeksom 12.

T	R	d	e	č	a		z	o	r	a		-		m	o	k	r	a		g	o	r	a	.
P					s	k	u	p	a	j		j	e	m	o									
j					0	1	2	3	4	5	6	7	8	9	10									
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23

Korak 1a, 1b, 1c:
Primerjamo zadnji znak iz vzorca P, znak 'o' z znakom 'o' iz besedila T. Znaka sta enaka. Zmanjšamo indeksa i in j za ena. Nastopi korak 1b, kjer primerjamo znak 'm' iz vzorca P z znakom 'm' iz besedila T. Znaka sta enaka. Zmanjšamo i in j za ena in nastopi korak 1c. Primerjamo znak 'e' iz P z znakom ' ' iz T. Znaka nista enaka, zato pogledamo, ali vzorec P vsebuje znak ' '. Ker ga vsebuje na mestu z indeksom 6, celoten vzorec P premaknemo tako, da se v P znak na mestu 6 poravna z znakom v T na mestu 12.

T	R	d	e	č	a		z	o	r	a		-		m	o	k	r	a		g	o	r	a	.
P							s	k	u	p	a	j		j	e	m	o							
j							0	1	2	3	4	5	6	7	8	9	10							
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23

Korak 2:
Zadnji znak iz vzorca P je enak znaku 'o'. Pogledamo ali je enak znaku 'r' iz besedila T. Znaka nista enaka. Vzorec P ne vsebuje znaka 'r', zato celoten P premaknemo tako, da se znak na mestu P[0] poravna z znakom na mestu T[17].

T	…	m	o	k	r	a		g	o	r	a	.				
P						s	k	u	p	a	j		j	e	m	o
j						0	1	2	3	4	5	6	7	8	9	10
i	…	13	14	15	16	17	18	19	20	21	22	23				

Dolžina preostanka besedila T je manjša od dolžine besedila P, zato zaključimo s primerjanji. Algoritem v tem primeru vrne vrednost -1, kar pomeni, da ujemanja nismo našli. Torej v besedilu T ne obstaja tak podniz, ki se znak po znak ujema z vzorcem P.

1.2.2 Algoritem

Algoritem bo sestavljen iz treh metod, `algoritemBoyerMoore(String t, String p)`, `zadnji(String p)` in `abeceda(String t)`.

```
import java.util.*;
public class TekstovniAlgoritmi
{
  public static int algoritemBoyerMoore(String t, String p)
  {
    //implementacija algoritma Boyer Moore
  }

  private static Hashtable zadnji(String p, Hashtable abeceda)
  {
    //implementacija funkcije L(x)
  }

  private static Hashtable abeceda(String t)
  {
    //gradimo abecedo, ki predstavlja nabor znakov, ki jih vsebuje
    //besedilo t
  }
}
```

Najprej si oglejmo metodo `abeceda(String t)`. Metoda bo kot vhodni parameter dobila besedilo `t`, ki je tipa `String`, vrnila pa bo slovar, ki je tipa `Hashtable`. Slovar bo vseboval vse znake, ki so vsebovani v besedilu `t`. Ena od prednosti uporabe tipa `Hashtable` je, da ni potrebno posebej skrbeti za podvojene znake, saj v objektu tipa `Hashtable` ne moremo imeti podvojenih znakov (ključev). Torej, če besedilo T vsebuje dva znaka 'a', se bo v slovar vpisal le eden. Znaki bodo ključi slovarja. Vrednosti ključev bomo uporabili pri gradnji funkcije L, zato jim v tej metodi določimo vrednosti -1. Ko bomo z metodo `zadnji` gradili funkcijo L(x), bodo določeni ključi (tisti znaki, ki bodo nastopali v vzorcu P) spremenili vrednost na ustrezno, ostali pa bodo (kot zahteva fukcija L) ostali na -1. Oglejmo si metodo.

```
private static Hashtable abeceda(String t)
{
  //naredimo nov slovar, ki je na začetku prazen
  Hashtable abeceda = new Hashtable();
  //sprehodimo se po besedilu t
  for(int i = 0; i < t.length(); i++)
  {
    //vstavimo znak v slovar abeceda
   abeceda.put(t.charAt(i), -1);
  }
  return abeceda;
}
```

S pomočjo metode `zadnji(String p, Hashtable abeceda)` bomo zgradili funkcijo `L(x)`, ki nam bo povedala, na katerem mestu dani vzorec `p` vsebuje znak `x` iz abecede A. Naredili bomo nov objekt `lx` tipa `Hashtable`, ki bo klon slovarja `abeceda`. S tem bomo slovar abeceda ohranili nespremenjen, če ga bomo v nadaljevanju še potrebovali. Če bo vzorec `p` vseboval znak `x`, bomo v slovar `lx` zapisali indeks, ki bo označeval, kje se v vzorcu `p` nahaja znak `x`. Če vzorec `p` ne bo vseboval znaka `x`, bomo vrednost za ključ `x` pustili nespremenjeno, torej `-1`. Kot vhodna parametra vzamemo iskani vzorec `p`, ki je tipa `String` in slovar `abeceda`, ki je tipa `Hashtable`. Kot izhodni parameter bo metoda vrnila na novo narejeni slovar `lx`, ki je tipa `Hashtable`.
Ko smo s pomočjo metode `abeceda(String t)` gradili slovar `abeceda`, smo znakom, ki predstavljajo ključ slovarja, dodelili vrednost `-1`. Tako nam v metodi `zadnji(String p, Hashtable abeceda)` ne bo potrebno preverjati, ali vzorec `p` vsebuje znak iz slovarja abeceda, ampak bomo za vse znake, ki jih `p` vsebuje, le vpisali indeks, ki bo označeval, kje se znak iz slovarja abeceda nahaja v vzorcu `p`. Ker bomo šli od leve proti desni, bomo v primeru, ko v vzorcu isti znak nastopa večkrat, na koncu torej imeli indeks

zadnjega med enakimi znaki. To pa je tisto, kar zahteva funkcija L. Če bo vzorec p vseboval tudi znake, ki niso v slovarju abeceda, jih bomo dodali, čeprav jih ne bomo nikoli uporabili.

```
private static Hashtable zadnji(String p, Hashtable abeceda)
{
  //skopiramo vrednosti, ključe iz Hashtable abeceda v nov objekt
  //tipa Hashtable
  Hashtable lx = (Hashtable)abeceda.clone();
  //sprehodimo se po vzorcu p
  for(int i = 0; i < p.length(); i++)
  {
    //V slovar dodamo ključ,ki je enak znaku na mestu i v vzorcu p.
   //če dodamo nov element in ta podvaja ključ že obstoječega
   //elementa, se bo ključ zamenjal
    //vrednost ključa je celo število, ki označuje
    //mesto kjer se znak pojavi v vzorcu p.
    lx.put(p.charAt(i), i);
  }
  return lx;
}
```

Opisali smo pomožne metode, ki jih bomo uporabili v metodi `algoritemBoyerMoore(String t, String p)`. Metoda bo kot vhodna prametra dobila besedilo `t` in vzorec `p`, ki sta tipa `String`. Vrnila bo celo število, tipa `int`, ki bo označevalo začetno mesto ujemanja vzorca `p` v besedilu `t`.
S pomočjo indeksa `i` se bomo sprehajali po besedilu `t`. Ker algoritem Boyer-Moore uporablja tehniko zrcalo, bo indeks `i` na začetku enak `m - 1`, kjer je `m` dolžina vzorca `p`. Na začetku bomo preverili, ali je dolžina vzorca `p` večja od dolžine besedila `t`. Če bo dolžina vzorca `p` večja od dolžine besedila `t`, bomo algoritem zaključili in vrnili `-1`, sicer bomo začeli s preverjanji. Dokler ne pridemo do konca besedila `t`, preverjamo, ali se `j-ti` znak v `p` ujema z `i-tim` znakom v `t`. Če sta znaka enaka, uporabimo tehniko zrcalo, sicer uporabimo tehniko preskakovanja znakov.

```
public static int algoritemBoyerMoore(String t, String p)
{
  //pokličemo metodo, ki nam zgradi slovar abeceda
  Hashtable abeceda = abeceda(t);
  //pokličemo metodo, ki nam zgradi funkcijo L(x)
  Hashtable l = zadnji(p, abeceda);
  int n = t.length(); //dolžina danega besedila T
  int m = p.length(); //dolžina iskanega vzorca P

  int i = m-1; //z indeksom i se bomo sprehajali po besedilu T
               // začnemo tako desno, kot je dolg vzorec P

  //preverimo ali je dolžina iskanega vzorec P večja od dolžine
  // besedila T
  if(i > n-1) return -1; // vzorca zagotovo ni v besedilu, saj je predolg
  //z indeksom j se premikamo od zadnjega znaka proti prvemu znaku v
  //vzorcu P
  int j = m-1;
  do //iskanje ujemanja ponavljamo, dokler ne pridemo do konca besedila T
  {
    if(p.charAt(j) == t.charAt(i)) //pogledamo, ali se znaka ujemata
    {
      //če pridemo do konca vzorca p (j==0) smo našli ujemanje
      //in vrnemo začetni indeks ujemanja
      if (j == 0) return i;
      //sicer uporabimo tehniko zrcalo (looking-glass)
      i--;
      j--;
    }
    //znaka nista enaka, zato uporabimo tehniko preskakovanja znakov
    else
    {
      //pogledamo vrednost L(x) za trenutni znak iz besedila t
      int lx = Integer.parseInt(l.get(t.charAt(i)).toString());
      //premik vzorca p
      i = i + m - Math.min(j, 1 + lx); // nov položaj v T
      j = m - 1; // ponovno začnemo pri zadnjem znaku v vzorcu P
    }
  }while(i <= n-1);
  return -1; //ujemanja nismo našli, zato vrnemo -1
}
```

1.2.3 Časovna zahtevnost

Oglejmo si časovno zahtevnost algoritma.
Velikost problema, ki ga obravnavamo, je odvisna od dolžine obeh nizov, ki sta vhodna parametra. Kot vhodna parametra algoritem sprejme besedilo T in vzorec P. Dolžina T, ki jo označimo z `n` in P, ki jo označimo z `m`, predstavljata velikost problema. Za algoritem Boyer-Moore, ki ga označimo z `b`, določimo časovno zahtevnost problema $T_b(n, m)$.

Opišimo grob premislek, kako je s časovno zahtevnostjo. Bralec, ki potrebuje natančno analizo, jo bo našel npr. v knjigi Goodrich, M. T., Tamassia, R., *Data structures and algorithms in java*.
Algoritem kliče metodi `abeceda(t)` in `zadnji(p, abeceda)`. V metodi `abeceda(t)` se največ `n`-krat sprehodimo po besedilu `t`. Torej porabi največ `O(n)` časa, da se izvede. V metodi `zadnji(p, abeceda)` se največ `m`-krat sprehodimo po vzorcu `p`, zato metoda porabi največ `O(m)` časa za izvajanje. Algoritem je sestavljen še iz zanke `do-while` v kateri se s pomočjo indeksa `i`, ki teče od `m - 1` do vključno `n - 1`, sprehodimo po besedilu T. Dokler je še kaj znakov v besedilu T, primerjamo, ali vzorec P vsebuje trenutni znak iz besedila T. Tu opravimo največ `n * m` primerjanj. Časovna zahtevnost algoritma je

seštevek primerjanj, ki jih izvedemo, ko gradimo abecedo, primerjanj, ko gradimo funkcijo `L(x)` in primerjanj v zanki `do-while`. To je `O(n) + O(m) + O(n * m)`. Torej bo `O(n * m)`.

Oglejmo si kakšna je časovna zahtevnost algoritma v najboljšem, pričakovanem in najslabšem primeru.

Najboljši primer
Če zanemarimo očitno nesmiselen primer, ko je dolžina vzorca P daljša od dolžine besedila T, se v najboljšem primeru vzorec P nahaja na začetku besedila T. Takrat dolžina besedila T ne vpliva na časovno zahtevnost. Za vsak znak v vzorcu P opravimo eno primerjavo, torej je število primerjanj enako `m`.

$$T_b(n, m) = O(m)$$

Najslabši primer
Največ dela bomo imeli, če vzorca P v besedilu T sploh ni. Pri tem bo najslabši primer nastopil takrat, ko bomo pri preverjanju v notranji podzanki vedno prišli prav do konca. Premislek nam pokaže, da bo tak najslabši par denimo "xxxxxxxxz" in "xxy". Tu nam tudi L(x) nič ne pomaga, saj se bomo vedno premikali le za en znak naprej.
Pokažimo, da formula za premik pove, da je premik 1.
Vrednosti L(x) so enake:

A	x	z
L(x)	1	-1

Zapišimo niza v tabelo:

A	x	x	x	x	x	x	x	x	z
L(x)	x	x	y						

Primerjanja začnemo od zadaj in vidimo, da se znaka 'x' in 'y' ne ujemata. Poglejmo kaj pravi formula [BM1]

```
i = it + m - min(j, 1 + L(T[it])) = 2 + 3 - min(2, 1 + 1) = 3,
```

kar nam pove, da vzorec P premaknemo tako, da se znak na mestu P[2] ujema z znakom na mestu T[3].

A	x	x	x	x	x	x	x	x	z
L(x)		x	x	y					

Do podobne ugotovitve pridemo vse do zadnjega koraka.

V tem primeru se izvede maksimalno število primerjanj.

$$T_b(n, m) = O(n * m)$$

Predstavili smo zelo poenostavljeno različico algoritma Boyer-Moore. Pod imenom Boyer-Moore pa pogosto srečamo izboljšano različico, ki je časovne zahtevnosti `O(n + m + |A|)`. Tu je `|A|` dolžina abecede. Izboljšavo doseže s posebno tehniko preskakovanja znakov. Prihranek doseže s tem, da si zapomni del niza iz vzorca P, ki ustreza trenutno najdenemu ujemanju vzorca P v besedilu T. Tako lahko vzorec P premakne za več mest v desno. Različice algoritma Boyer-Moore, ki uporabljajo to tehniko, so na primer algoritem Turbo Boyer-Moore ali algoritem Boyer-Moore-Horspool.

Pričakovana časovna zahtevnost
Ker je analiza pričakovane časovne zahtevnosti zapletena, še posebej zaradi uporabe tehnike preskakovanja, povejmo le, da je pričakovana časovna zahtevnost enaka `O(n * m)`.

Kadar je dolžina niza P sorazmerna dolžini besedila T, torej $m \approx n$, je metoda kvadratične časovne zahtevnosti

$$T_b(n, m) = O(n * m) = O(n * (n)) = O(n^2)$$

1.3 Algoritem Knuth-Morris-Pratt

Algoritem Knuth-Morris-Pratt je po logiki iskanja podoben algoritmu Groba Sila. Ujemanje vzorca P v danem besedilu T išče od leve proti desni. Uporablja pa posebno tehniko iskanja in je zato hitrejši od algoritmov Groba Sila in Boyer-Moore. Že opisana algoritma iščeta ujemanje vzorca P v danem besedilu T in če se na določenem koraku znaka ne ujemata, algoritma zavržeta vse pridobljene informacije in začneta znova preverjati, ali T vsebuje vzorec P. Algoritem Knuth-Morris-Pratt pa poskusi to pridobljeno informacijo izkoristiti s pomočjo tako imenovane funkcije neuspeha (Failure Function). Ta nam pove, za koliko se lahko premaknemo, ne da bi izgubili kako ujemanje.

Denimo, da imamo dano besedilo T = "aababababc". Iščemo, ali besedilo T vsebuje tako podzaporedje znakov, da se znak po znaku ujema z vzorcem P = "ababc". Spomnimo se, kako izvedemo tako iskanje s pomočjo algoritma Groba Sila. Nato si bomo ogledali, kako iskanje izvede algoritem Knuth-Morris-Pratt.

T	a	a	b	a	b	a	b	c
P	a	b	a	b	c			
j	0	1	2	3	4			
i	0	1	2	3	4	5	6	7

Algoritem Groba Sila išče ujemanje vzorca P v besedilu T tako, da preverja vsak znak posebej.

Korak 1a, 1b:
Na prvem koraku vzamemo prvi znak iz besedila T, znak 'a' in ga primerjamo s prvim znakom iz vzorca P, znakom 'a'. Znaka sta enaka. Povečamo indeksa i in j za ena. V naslednjem koraku 1b primerjamo, ali sta znaka na mestu T[1] ('a') in P[1] ('b') enaka. Znaka nista enaka. Celoten vzorec P premaknemo za en znak v desno tako, da na naslednjem koraku primerjamo znak na mestu T[1] z znakom na mestu P[0].

T	a	a	b	a	b	a	b	c
P		a	b	a	b	c		
j		0	1	2	3	4		
i	0	1	2	3	4	5	6	7

Korak 2a do 2e:
Naslednja dva znaka, ki jih primerjamo, sta znaka T[1], znak 'a' in P[0], znak 'a'. Znaka sta enaka. Povečamo indeksa i in j za ena. Na koraku 2b primerjamo, ali je znak 'b' (T[2]) enak znaku 'b' (P[1]). Znaka sta enaka. Povečamo indeksa i in j za ena in primerjamo znak T[3] ('a') z znakom P[2] ('a'). Znaka sta enaka. Tudi naslednja znaka T[4] ('b') in P[3] ('b') sta enaka. Spet povečamo indeksa i in j za ena. Pogledamo, ali je znak v T na mestu 5 enak znaku P[4]. Prvi znak je 'a', drugi pa 'c', znaka nista enaka. Celoten vzorec P premaknemo za en znak v desno tako, da v naslednjem koraku primerjamo znak na mestu T[2] z znakom na mestu P[0].

T	a	a	b	a	b	a	b	c
P			a	b	a	b	c	
j			0	1	2	3	4	
i	0	1	2	3	4	5	6	7

Na koraku 2e primerjamo, ali se znak iz besedila T, ki je na mestu z indeksom i = 4, torej T[4], ujema z znakom iz vzorca P na mestu z indeksom j = 3, torej P[3]. Ugotovimo, da znaka nista enaka. Sedaj pridemo do vprašanja, kako daleč naj premaknemo vzorec P. Algoritem Groba Sila premakne celoten vzorec P za en znak v desno. To pri opisanem primeru pomeni, da smo zavrgli vse informacije, ki smo jih na koraku 2 pridobili. Namreč na koraku od 2a do 2e smo ugotovili, da velja P[0] = T[1], P[1] = T[2], P[2] = T[3], znak P[3] ≠ T[4]. In algoritem GrobaSila v naslednjem koraku primerja znak na mestu T[2] z znakom na mestu P[0].
Tak način je sprejemljiv, če sta besedilo T in vzorec P kratka niza, sicer tako iskanje porabi precej časa. Zato uporabimo boljši algoritem, ki zna shraniti pridobljene informacije. To je algoritem Knuth-Morris-Pratt.

Oglejmo si, kako rešimo zgornji primer s pomočjo algoritma Knuth-Morris-Pratt. Na začetku poglavja smo omenili, da algoritem uporablja funkcijo neuspeha. Zapišimo funkcijo neuspeha F(k) za dani vzorec P = "ababc". Kako to funkcijo neuspeha za dani vzorec P zgradimo, si bomo ogledali v nadaljevanju.

j	0	1	2	3	4
P[j]	a	b	a	b	c
k	0	1	2	3	4
F(k)	0	0	1	2	0

Vpišimo niza T in P v tabelo:

T	a	a	b	a	b	a	b	c
P	a	b	a	b	c			
j	0	1	2	3	4			
i	0	1	2	3	4	5	6	7

Korak 1a, 1b:
Na prvem koraku vzamemo prvi znak iz besedila T, znak 'a' in ga primerjamo s prvim znakom iz vzorca P, znakom 'a'. Znaka sta enaka. Povečamo indeksa i in j za ena. V naslednjem koraku 1b primerjamo, ali sta znaka T[1] ('a') in P[1] ('b') enaka. Znaka nista enaka. Vemo, da smo zadnje ujemanje znakov našli na mestu P[0]. Zato je k = 0. Funkcija F(k) za k = 0 je enaka 0, torej F(0) = 0. Ker je F(0) = 0, se s celotnim vzorcem P premaknemo za en znak v desno.

T	a	a	b	a	b	a	b	c
P		a	b	a	b	c		
j		0	1	2	3	4		
i	0	1	2	3	4	5	6	7

Korak 2a do 2e:
Naslednja dva znaka, ki ju primerjamo, sta znaka T[1] in P[0]. Ker sta obakrat to znaka 'a', nadaljujemo. Ugotovimo, da se ujemata tudi T[2] in P[1], T[3] in P[2] ter T[3] in P[2]. Ko primerjamo T[5] in P[4], ustrezna znaka ('a' in 'c') nista enaka. Zadnje ujemanje smo v P našli na mestu 3. Zato je k = 3. Funkcija F(k) za k = 3 je enaka 2, torej F(3) = 2. Neujemanje je v T nastopilo na mestu z indeksom it = 5, torej ko smo primerjali znak T[5]. Ker je F(3) = 2, bo novi indeks i = it – F(k) = 5 - 2 = 3. Na naslednjem koraku bomo primerjali znak na mestu T[3] z znakom na mestu P[0].

T	a	a	b	a	b	a	b	c
P				a	b	a	b	c
j				0	1	2	3	4
i	0	1	2	3	4	5	6	7

1.3.1 Funkcija neuspeha ali F(k)

Funkcijo F(k) zgradimo za iskani vzorec P. Pove nam, za koliko mest lahko premaknemo vzorec P in s tem ne preskočimo nobene pojavitve iskanega vzorca v besedilu T.

Denimo, da smo na določenem koraku prišli do naslednjega stanja

P[0] = T[it], P[1] = T[it + 1], …, P[k] = T[it + k] in P[k + 1] ≠ T[it + k + 1],

Sedaj pogledamo vrednost funkcije neuspeha (F) pri k. Na podlagi tega bomo vedeli, za koliko lahko premaknemo vzorec P.

Na primeru si oglejmo, kako F(k) vpliva na premik vzorca P.
Naj bo T = "mamina" in P = "mama". Najprej zapišimo vrednosti funkcije F(k).

j	0	1	2	3
P	m	a	m	a
k	0	1	2	3
F(k)	0	0	1	2

Zapišimo niza T in P v tabelo.

T	m	a	m	i	n	a
P	m	a	m	a		

Razvidno je, da velja naslednje: P[0] = T[0], P[1] = T[1], P[2] = T[2] in P[3] ≠ T[3]. Označimo mesto neujemanja.

T	m	a	m	i	n	a
P	m	a	m	a		

S celotnim vzorcem P se premaknemo tako, da poravnamo znak na mestu T[3] z znakom na mestu P[0].

T	m	a	m	i	n	a	
P				m	a	m	a

Sedaj nastopi F(k), ki nam pove, za koliko mest preveč smo premaknili vzorec P. Zadnje ujemanje smo našli na mestu P[2]. Zato pogledamo F(2), ki ima vrednost 1. To pomeni, da se bomo z vzorcem P premaknili za eno mesto manj v levo, torej bi morali premik celotnega vzorca narediti za eno mesto manj v desno.

T	m	a	m	i	n	a
P			m	a	m	a

Vrednosti funkcije F so odvisne od pojavitve podniza v vzorcu P.
Vzemimo vzorec "mrak". Vrednosti funkcije F(k) za dani vzorec bodo vse enake 0, saj se v vzorcu vsak znak ponovi le enkrat. Za dani vzorec "grog" bodo vrednosti F(k) za k = 0, 1, 2 enake 0, za k = 3 pa bo F(3) enak 1. To pa zato, ker se znak 'g' v vzorcu ponovi dvakrat. V tem primeru je podniz enak "g". Dolžina podniza je 1, zato je F(3) enak 1.

Poglejmo si podrobneje idejo, kako dobimo vrednosti funkcije neuspeha za vzorec "aabaa".
F(0) bo enak 0, saj se znak 'a' pojavi prvič v vzorcu P. Iz tega sledi, da bo za vsak vzorec P vrednost F(0) enaka 0.

Sedaj računamo vrednost F za drugi znak iz vzorca P, znak 'a'. Znak 'a' se pojavi že na začetku vzorca P, torej se do sedaj pojavi dvakrat. Iz tega sledi, da smo našli podniz, ki je enak "a". Ker je dolžina podniza enaka ena, je F(1) = 1.
Vrednost F(2) je enaka 0, saj se znak 'b' prvič pojavi v vzorcu P.
Razmislek, kako izračunamo F(3), je podoben razmisleku, ki smo ga uporabili pri izračunu F(1). Znak 'a' se do sedaj ponovi trikrat. Pogledamo ali se je niz "ba" že pojavil v vzorcu P. Ker se niz "ba" še ni pojavil v vzorcu P, je podniz enak "a". Dolžina podniza je ena, zato je F(3) = 1.
Zadnja vrednost funkcije F bo enaka 2, poglejmo zakaj. V tem primeru vidimo, da se znak 'a' v vzorcu P pojavi štirikrat. Pogledati moramo tudi, če se je niz "aa" že pojavil v vzorcu P. Niz "aa" se je v vzorcu P že pojavil. Sedaj pogledamo ali se je niz "baa" že pojavil v vzorcu P. Opazimo, da se ni. Zato je podniz v tem primeru enak "aa". Dolžina podniza je 2, zato je F(4) = 2.

Poglejmo si še en primer, na katerem bomo videli, kaj vrednosti F(k) pomenijo.
Za vzorec P = "abcabcd" bodo vrednosti F(k) enake

j	0	1	2	3	4	5	6
P	a	b	c	a	b	c	d
k	0	1	2	3	4	5	6
F(k)	0	0	0	1	2	3	0

Poglejmo kaj pomenijo določene vrednosti F(k).
Vrednost F(2) = 0 nam pove, da se znak 'c' v vzorcu P, od mesta P[0] do mesta P[2], pojavi prvič.
F(3) = 1 nam pove, da smo v vzorcu P našli podniz, ki je enak "a" in je dolžine 1. Razmislimo, če to drži. Znak 'a' se od mesta P[0] do P[3] pojavi dvakrat. Označimo pojavitev znaka 'a' v vzorcu P.

j	0	1	2	3	4	5	6
P	a	b	c	a	b	c	d

To pomeni, da smo našli podniz "a", ki se je že pojavil v vzorcu P. Dolžina tega podniza je enaka ena. Zato je F(3) enak 1. Vidimo, da se naš razmislek ujema z izračunom.
Vzemimo še eno vrednost iz funkcije F, vrednost F(5) = 3 in poglejmo, kaj nam ta vrednost pove. Vrednost nam pove, da smo v vzorcu P našli podniz dolžine tri, ki se dvakrat ponovi v vzorcu P, od mesta P[0] do P[5]. Označimo ta podniz.

j	0	1	2	3	4	5	6
P	a	b	c	a	b	c	d

Oglejmo si, kaj moramo upoštevati pri računanju F(k).

Ko bomo gradili F(k), se bomo s spremenljivkama j in k sprehajali po vzorcu P in primerjali znak na mestu P[j] z znakom na mestu P[k].
Najprej prikažimo idejo, kako se bomo z i in k sprehajali po vzorcu P.

j	0	1	2	3	4
P	a	a	b	a	a
k	0	1	2	3	4
P	a	a	b	a	a

Ker iz spodnjih pravil sledi, da je F(0) vedno enak 0, bomo primerjanja dejansko začeli pri k = 1 in j = 0.

j	0	1	2	3	4
P	a	a	b	a	a
k	0	1	2	3	4
P	a	a	b	a	a

Ker sta znaka enaka, povečamo j in k za ena.

j	0	1	2	3	4
P	a	a	b	a	a
k	0	1	2	3	4
P	a	a	b	a	a

Znaka nista enaka, zato bomo na naslednjem koraku primerjali znaka na mestu j = F(j - 1) = F(1 - 1) = 0 in k = 2.

j	0	1	2	3	4
P	a	a	b	a	a
k	0	1	2	3	4
P	a	a	b	a	a

Znaka nista enaka. Ker je j = 0, povečamo k za ena. Na naslednjem koraku primerjamo znak na mestu j = 0 z znakom na mestu k = 3.

Postopek nadaljujemo, dokler ne pridemo s spremenljivko k do konca vzorca P. Iz danega primera je razvidno, da se bomo s spremenljivko j premikali naprej in nazaj po vzorcu P, medtem ko se bomo s spremenljivko k premikali samo naprej.

Pri določanju vrednosti F(k) bomo upoštevali naslednja pravila:

Pravilo 1 (P1):
F(0) je za poljuben vzorec P vedno enak 0.
Pravilo 2 (P2):
Če sta znaka na mestu P[j] in P[k] enaka, je F(k) = j + 1. Ko uporabimo to pravilo, pri gradnji povečamo oba indeksa k in j za 1. Na naslednjem koraku bomo torej primerjali znak P[j + 1] z znakom P[k + 1].
Pravilo 3 (P3):
Če znaka na mestu P[j] in P[k] nista enaka, pogledamo vrednost indeksa j. Če je j > 0, postopamo kot pravi pravilo 4. Če pa je j <= 0, je vrednost F(k) = 0. Za nadaljevanje indeks j nastavimo na 0 in na naslednjem koraku primerjamo znak P[0] z znakom P[k + 1].
Pravilo 4 (P4):
Znaka P[j] in P[k] se ne ujemata in j > 0. Vrednost spremenljivke j bo F(j - 1), torej bomo na naslednjem koraku primerjali P[F(j - 1)] s P[k].

Oglejmo si nekaj primerov, kako zgradimo funkcijo F(k).

Primer:
Dan imamo vzorec P = "ababc". Računamo funkcijo F(k), za k = 0, …, 4. Zapišimo rezultate v tabelo in si oglejmo postopek računanja.

j	0	1	2	3	4
P	a	b	a	b	c
k	0	1	2	3	4
F(k)	0	0	1	2	0

Korak 0:
Nastavimo j = 0, k = 0 in F[0] = 0 (P1).
Korak 1:
Računamo F(1), torej k = 1 in j = 0. Primerjamo znaka P[1] ('b') in P[0] ('a'). Ker sta znaka različna in je j = 0, je prišlo do možnosti 3. Upoštevamo **P3** in iz tega sledi F(1) = 0. Indeks j je 0 in k je sedaj 2.
Korak 2:
Pri k = 2 in j = 0 primerjamo znaka P[2] ('a') in P[0] ('a'). Znaka se ujemata in iz **P2** sledi F(2) = j + 1 = 1. Povečamo j za ena, j = 1. Na naslednjem koraku primerjamo znaka na mestu P[3] in P[1].
Korak 3:
Pri k = 3 in j = 1 primerjamo znaka P[3] ('b') in P[1] ('b'). Znaka se ujemata in iz **P2** sledi F(3) = j + 1 = 2. Povečamo j za ena, j = 2. Na naslednjem koraku primerjamo znaka na mestu P[4] in P[2].
Korak 4:
Pri k = 4 in j = 2 primerjamo znaka P[4] ('c') in P[2] ('a'). Znaka nista enaka. Ker je j > 0, iz **P4** sledi, da na naslednjem koraku primerjamo znaka pri k = 4 in j = F(j - 1) = F(1) = 0.
Korak 5:
Sedaj pri k = 4 in j = 0 primerjamo znaka 'c' in 'a'. Znaka nista enaka. Ker je j = 0, iz **P3** sledi F(4) = 0.
Ker smo s tem določili vrednost F(k) za vse k, je postopek končan.

Vrednosti 0, 0, 1, 2, 0 nam povejo, da vzorec P vsebuje dva podniza, ki se v vzorcu P ponovita dvakrat in sta dolžine 1 in 2. To sta podniza "a" in "ab".

Oglejmo si še, kako funkcijo neujemanja izračunamo še za vzorce P = "mama", P = "abacab" in P = "sadje", ki jih bomo uporabljali v nadaljevanju.

Najprej poglejmo zadnji vzorec, torej P = "sadje". Če malo premislimo, ugotovimo, da je za vzorce, ki imajo same različne znake, vrednost funkcije F povsod 0. Pri vsakem k bo vedno izpolnjen P3 in s tem F(k) = 0.

j	0	1	2	3	4
P	s	a	d	j	e
k	0	1	2	3	4
F(k)	0	0	0	0	0

Za vzorec P = "mama" bo tabela enaka:

j	0	1	2	3
P	m	a	m	a
k	0	1	2	3
F(k)	0	0	1	2

Oglejmo si, kako smo izračunali vrednosti F(k).

Korak 0:
Pri k = 0 je vrednost F(k) = 0.
Korak 1:
Pri k = 1 in j = 0 preverjamo, ali sta znaka 'a' in 'm' enaka. Ker znaka nista enaka in je j = 0, upoštevamo pravilo P3 in iz tega sledi F(1) = 0. k povečamo za ena.

Korak 2:
Pri k = 2 in j = 0 primerjamo znaka 'm'. Ker sta znaka enaka, iz možnosti P2 sledi F(2) = j + 1 = 1. Indeksa k in j povečamo za ena.
Korak 3:
Pri k = 3 in j = 1 primerjamo znaka 'a' in 'a'. Ker sta enaka, iz možnosti P2 sledi F(3) = j + 1 = 2.

V vzorcu P najdemo dva podniza, ki se ponovita dvakrat. To sta podniza "m" in "ma". Zato je vrednost F(2) = 1, kar nam pove, da je trenutno najdaljši podniz dolžine 1, to je m. F(3) = 2, kar pove, da je najdaljši podniz dolžine 2 in to je "ma".

Sedaj pa še vrednosti F(k) za P = "abacab":

Korak 0:
Pri k = 0 je vrednost F(k) = 0.
Korak 1:
Pri k = 1 in j = 0 preverjamo ali sta znaka 'a' in 'b' enaka. Ker znaka nista enaka in je j = 0, upoštevamo možnost P3 in iz tega sledi F(1) = 0. k povečamo za ena.
Korak 2:
Pri k = 2 in j = 0 primerjamo znaka 'a'. Znaka sta enaka. Iz možnosti P2 sledi F(2) = j + 1 = 1. Indeksa k in j povečamo za ena.
Korak 3:
Pri k = 3 in j = 1 primerjamo znaka 'c' in 'b'. Znaka nista enaka. Ker je j > 0, bomo na naslednjem koraku primerjali znaka na mestu k = 4 in j = F(j - 1) = F(0) = 0.
Korak 4:
Ker sta znaka ne mestu k = 4 in j = 0 enaka, je F(4) = j + 1 = 1.
Korak 5:
Znaka na mestu k = 5 in j = 1 sta enaka. Zato je F(5) = j + 1 = 2.

In tabela je torej

j	0	1	2	3	4	5
P	a	b	a	c	a	b
k	0	1	2	3	4	5
F(k)	0	0	1	0	1	2

Vrednost F(0) = 1 in F(4) = 1 nam povesta, da smo našli podniz v vzorcu P, ki se ponovi trikrat in je dolžine 1, to je "a". Najdaljši podniz, ki se ponovi dvakrat, je "ab", zato je F(5) = 2.

1.3.2 Algoritem Knuth-Morris-Pratt

Algoritem primerja dano besedilo T z iskanim vzorcem P. Vsakič, ko najdemo ujemanje (znak na i-tem mestu v T-ju se ujema z znakom na j-tem mestu v P-ju) povečamo trenutna indeksa i in j za ena. Če ujemanja ne najdemo, in smo predhodno napredovali v P (torej je j > 0), potem s pomočjo funkcije F(k) določimo, na katerem mestu v T moramo nadaljevati s primerjanji P s T. Proces ponavljamo, dokler ne najdemo podniza v besedilu T, ki se v celoti ujema z iskanim vzorcem P, ali pa dokler indeks v T-ju ne doseže dolžine T.

Oglejmo si na primerih, kako premikamo vzorec P.

Primer 1:
Pokažimo najprej, da se algoritem večkrat obnaša povsem enako kot algoritem Groba Sila. To se zgodi vedno, ko se prvi znak v nizu ne pojavi nikjer (razen morda ob ujemanju) v besedilu T.

Poglejmo zgled, ko prvega znaka v besedilu ni.
Dano imamo besedilo T = "kladivo". Iščemo, ali v besedilu T obstaja tak podniz, ki je enak vzorcu P = "sadje". Zapišimo niza v tabelo.

T	k	l	a	d	i	v	o
P	s	a	d	j	e		

Pogledamo, ali je znak na mestu P[0] enak znaku na mestu T[0]. Znaka nista enaka. Funkcija F za niz "sadje" (no, dejansko to velja za poljuben niz) je taka, da velja F(0) = 0, saj popravka v levo ne more biti. S celotnim vzorcem P se zato premaknemo za en znak v desno.

T	k	l	a	d	i	v	o
P		s	a	d	j	e	

Tudi v naslednjem koraku ni ujemanja (saj smo rekli, da znaka 's' v besedilu ni). V tem primeru se bomo premikali na isti način kot pri algoritmu Groba Sila, saj algoritem Knuth-Morris-Pratt prinaša prednosti pri premikanju le, če uspemo najti vsaj delno uejmanje.

Tudi če vzorec je v besedilu (a se noben drug del besedila ne začne s prvo črko iz P) se zgodi podobno.
Recimo sedaj, da imamo dano besedilo T = "kopirni stroj" in vzorec P = "stroj". Po nekaj korakih pridemo do stanja:

T	k	o	p	i	r	n	i		s	t	r	o	j
P			s	t	r	o	j						

Primerjamo znak 'p' na mestu T[2] z znakom 's' na mestu P[0]. Znaka nista enaka, zato celoten vzorec P premaknemo za en znak v desno.

T	k	o	p	i	r	n	i		s	t	r	o	j
P				s	t	r	o	j					

Če se torej že prvi znak v P ne ujema z nobenim v T, se premikamo po en znak v desno.

Primer 2:
Poglejmo primer, ko bi prevelik premik v desno pripeljal do morebitne izgube rešitve.

Denimo, da se vzorec P ujema z besedilom T v k znakih, torej P[0 … k] = T[i … i + k], na k + 1 znaku pa se niza ne ujemata, torej P[k + 1] ≠ T[i + k + 1]. Zadnje ujemanje smo torej našli na k-tem mestu. Pogledamo vrednost funkcije F(k). Če je F(k) > 0, to pomeni, da smo v vzorcu P našli podniz, ki se vsaj dvakrat pojavi v vzorcu P. Zato se s celotnim vzorcem P premaknemo za k mest v levo, torej nazaj od zadnjega neujemanja. Na naslednjem koraku primerjamo, ali se znak na mestu P[0] ujema z znakom na mestu T[it - F(k)]. Tu je it indeks, kjer smo našli neujemanje znaka iz vzorca P v besedilu T, torej it = i + k + 1. Novi indeks i (torej mesto v besedilu T) je enak it - F(k).

Poglejmo si idejo še shematsko. Označimo mesto neujemanja z rdečim pravokotnikom.

i	…	i	i+1	i+2	…	i+k	i+k+1	…
T	…	A	A	A	…	A	A	…
P		A	A	A	…	A	D	
k		0	1	2	…	k	k+1	

Opazimo, da najdemo delno ujemanje, in sicer P[0…k] = T[i … i + k]. Ker znak na mestu P[k + 1] ni enak znaku na mestu T[i + k + 1], se z vzorcem P premaknemo za eno mesto v desno od mesta zadnjega ujemanja.

i	…	i	i+1	i+2	…	i+k	i+k+1		
T	…	A	A	A	…	A	A	…	
P							A	A	…
k							0	…	

S tem premikom bi verjetno izgubili rešitev. Zato pogledamo v funkcijo F, ki nam pove, za koliko mest preveč smo premaknili vzorec P.
Premislimo kako bi izračunali vrednost funckije neuspeha. Za začetek izračunamo vrednost F(1). Znak na mestu P[1] je enak 'A'. Ker se znak pojavi tudi na mestu P[0] pomeni, da smo našli podniz, ki se dvakrat ponovi v nizu P od mesta P[0] do P[1]. Ta podniz je enak "A". Ker je dolžina podniza enaka ena, bo vrednost F(1) = 1. Sedaj izračunamo vrednost F(2). Znak pri F(2) je tudi enak 'A'. Pojavi se na prvih treh mestih v P. Sedaj preverimo ali se je niz "AA" že pojavil v vzorcu P od mesta P[0] do P[2]. Vidimo, da se je že pojavil (P[0 1] = "AA"). Sedaj pa izračunamo vrednost pri F(k). Na mestu P[k] je znak enak 'A'. Sedaj preverjamo ali se je niz "AA … A" (teh A-jev je dejansko k) že pojavil v vzorcu P. Vidimo da se pojavi, in sicer od mesta P[0] do P[k - 1]. Dolžina tega niza je enaka k, zato je F(k) = k.
Iz razmisleka sledi, da se bomo z vzorcem P premaknili za k mest nazaj.

i		…	i	i+1	i+2	…	i+k	i+k+1	…
T	…	A	A	A	…	A	B	…	
P			A	A	A	…	A	D	
k			0	1	2	…	k	k+1	

Oglejmo si še en zgled, kjer bi premik v desno pripeljal do izgube rešitve.
Dano imamo besedilo T = "mamina potica". Iščemo, ali v besedilu T obstaja tak podniz, ki je enak vzorcu P = "mama". Najprej zapišimo že prej izračunane vrednosti za F(k) za niz "mama".

j	0	1	2	3
P	m	a	m	a
k	0	1	2	3
F(k)	0	0	1	2

Zapišimo niza T in P v tabelo:

T	m	a	m	i	n	a		p	o	t	i	c	a
P	m	a	m	a									
j	0	1	2	3									
i	0	1	2	3	4	5	6	7	8	9	10	11	12

Prvi trije znaki iz vzorca P so enaki prvim trem znakom iz besedila T, torej velja P[0] = T[0], P[1] = T[1], P[2] = T[2]. Znaka P[3] ('i') in T[3] ('a') pa nista enaka. Zadnje ujemanje smo našli na mestu P[2].

Najprej razmislimo, kako bi lahko premaknili vzorec P.
S celotnim vzorcem P bi se premaknili tako, da bi poravnali prvi znak iz vzorca P, znak 'm', z znakom 'i' iz besedila T.

T	m	a	m	i	n	a		p	o	t	i	c	a
P				m	a	m	a						
j				0	1	2	3						
i	0	1	2	3	4	5	6	7	8	9	10	11	12

Pogledamo, za koliko mest preveč smo se premaknili. Zadnje ujemanje smo našli na mestu P[2]. Podniz "m" se v nizu P ponovi dvakrat, dolžina podniza je 1. Da ne spustimo rešitve, se moremo premakniti za en znak nazaj. Vzorec P bi torej premaknili tako, da bi ga postavili pod drugi m v T-ju.

T	m	a	m	i	n	a		p	o	t	i	c	a
P			m	a	m	a							
j			0	1	2	3							
i	0	1	2	3	4	5	6	7	8	9	10	11	12

Sedaj pa poglejmo, kaj o tem pravilu pravi zgoraj opisani postopek. Pogledamo vrednost F(k) za k = 2. Ker je F(2) = 1, se s celotnim vzorcem P premaknemo za T[3 - F(k)] mest v desno, torej bomo na naslednjem koraku primerjali znak na mestu T[3 - F(k)] = T[3 - 1] = T[2] z znakom na mestu P[0]. Postopek se torej ujema z našim razmislekom.

Primer 3:
Denimo, da se vzorec P ujema z besedilom T v k znakih, torej P[0 … k] = T[i … i + k] na k + 1 znaku pa se ne ujemata, torej P[k + 1] ≠ T[i + k + 1]. Torej smo zadnje ujemanje našli na k-tem mestu. Sedaj pogledamo vrednost funkcije F(k) za k. Če je F(k) = 0, se s celotnim vzorcem P premaknemo za eno mesto v desno od tam, kjer smo našli zadnje ujemanje. Tak premik lahko storimo zato, ker nam vrednost F(k) = 0 pove, da se znak na mestu P[k] prvič pojavi v vzorcu P. To posledično pomeni, da se ta znak ne pojavi v besedilu T od mesta T[i] do T[i + k]. Ker pa ne vemo ali se znak na mestu T[i + k +1] ujema z znakom na mestu P[0], bomo na naslednjem koraku primerjali, ali se znak na mestu P[0] ujema z znakom na mestu T[i + k + 1].

Dano imamo besedilo T = "sadovnjak in sadje". Iščemo ali v besedilu T obstaja tak podniz, ki je enak vzorcu P = "sadje". Najprej zapišimo izračunane vrednosti za F(k).

j	0	1	2	3	4
P	s	a	d	j	e
k	0	1	2	3	4
F(k)	0	0	0	0	0

Zapišimo niza T in P v tabelo:

T	s	a	d	o	v	n	j	a	k		i	n		s	a	d	j	e
P	s	a	d	j	e													
j	0	1	2	3	4													
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

Spet so prvi trije znaki iz vzorca P enaki prvim trem znakom iz besedila T, torej velja P[0] = T[0], P[1] = T[1], P[2] = T[2]. Znaka P[3] ('j') in T[3] ('o') nista enaka.

Spet najprej razmislimo, kako bi lahko premaknili vzorec P.
Opazimo, da se znak 'j' v vzorcu P pojavi enkrat, in sicer na mestu P[3]. To pomeni, da se znak 'j' ne more pojaviti v besedilu T od mesta T[0] do T[3]. Zato tudi ujemanja od mesta T[0] do T[3] ne moremo najti. Kar pomeni, da morebitno ujemanje vzorca P v besedilu T lahko najdemo od mesta T[3] naprej. Zato bi vzorec P premaknili tako, da bi njegov začetek postavili pod 'o' v T-ju.

T	s	a	d	o	v	n	j	a	k		i	n		s	a	d	j	e
P				s	a	d	j	e										
j				0	1	2	3	4										
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

In kaj pravi postopek? Zadnje ujemanje najdemo na mestu P[2]. Pogledamo vrednost F(k) za k = 2. Ker je F(2) = 0, se s celotnim vzorcem P premaknemo za eno mesto v desno od zadnjega mesta ujemanja, torej

T	s	a	d	o	v	n	j	a	k		i	n		s	a	d	j	e
P				s	a	d	j	e										
j				0	1	2	3	4										
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

1.3.3 Primer

Sedaj pa si opisano metodo oglejmo na celotnem primeru iskanja ujemanja.

V danem besedilu T = "abacaabaccabacabaabb" ugotovi, ali je P = "abacab" podniz besedila T.

V podpoglavju Funkcija neuspeha ali F(k) smo že naračunali ustrezno funkcijo F.

j	0	1	2	3	4	5
P	a	b	a	c	a	b
k	0	1	2	3	4	5
F(k)	0	0	1	0	1	2

Sedaj lahko začnemo z iskanjem vzorca P v danem besedilu T. Zapišimo oba niza v tabelo:

T	a	b	a	c	a	a	b	a	c	c	a	b	a	c	a	b	a	a	b	b
P	a	b	a	c	a	b														
j	0	1	2	3	4	5														
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Korak 0a do 0f:
Pogledamo znak na mestu P[0], znak 'a' in preverimo, ali se ujema z znakom na mestu T[0], znakom 'a'. Ujemanje obstaja, povečamo indeks i v T-ju in j P-ju.
Pogledamo znak na mestu P[1] ('b') in preverimo ali se ujema z znakom na mestu T[1] ('b'). Znaka se ujemata, zato povečamo indeks i v T-ju in j P-ju.
Znak na mestu P[2] ('a') se ujema z znakom na mestu T[2] ('a'), zato povečamo indeksa za ena.
Znak na mestu P[3] ('c') se ujema z znakom na mestu T[3] ('c'), zato povečamo indeksa za ena.
Znak na mestu P[4] ('a') se ujema z znakom na mestu T[4] ('a'), zato povečamo indeksa za ena.
Znak na mestu T[5] ('a') se ne ujema z znakom na mestu P[5] ('b'). Zadnje ujemanja najdemo na mestu P[4], kar pomeni, da je k = 4. Sedaj pogledamo funkcijo F in ugotovimo, da je F(4) = 1. Tako se z vzorcem P premaknemo za 5 - F(4) = 5 - 1 = 4 mesta v desno. V naslednjem koraku bomo primerjali znak na mestu T[4] z znakom na mestu P[0].

T	a	b	a	c	a	a	b	a	c	c	a	b	a	c	a	b	a	a	b	b
P					a	b	a	c	a	b										
j					0	1	2	3	4	5										
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Korak 1a, 1b:
Znak na mestu T[4] ('a') je enak znaku na mestu P[0] ('a'). Povečamo indeksa v nizih.
Znak na mestu T[5] ('a') se ne ujema z znakom na mestu P[1] ('b'). Zadnje ujemanje najdemo na mestu j = 0. Iz tega sledi, da je k = 0. Pogledamo v tabelo F(k) in ugotovimo, da je F(0) = 0. S celotnim vzorcem P se premaknemo za eno mesto v desno tako, da v naslednjem koraku primerjamo znak na mestu T[5] z znakom na mestu P[0].

T	a	b	a	c	a	a	b	a	c	c	a	b	a	c	a	b	a	a	b	b
P						a	b	a	c	a	b									
j						0	1	2	3	4	5									
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Korak 2a do 2e:
V prvih 4 primerjavah smo uspešni. Torej se znak na mestu P[0] ujema z znakom na mestu T[5], znak na mestu P[1] se ujema z znakom na mestu T[6], znak na mestu P[2] se ujema z znakom na mestu T[7] in znak na mestu P[3] se ujema z znakom na mestu T[8].
Na koraku 2 primerjamo, ali se znak na mestu T[9] ujema z znakom na mestu P[4]. Znaka se ne ujemata, zato pogledamo v tabelo F(k) in ugotovimo, da je F(3) = 0. Zato se z vzorcem P premaknemo za en znak v desno od zadnjega ujemanja, torej do T[9]

T	a	b	a	c	a	a	b	a	c	c	a	b	a	c	a	b	a	a	b	b
P										a	b	a	c	a	b					
j										0	1	2	3	4	5					
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Korak 3:
Znak na mestu T[9] ('c') se ne ujema z znakom na mestu P[0] ('a'). Povečamo indeks v besedilu T, z vzorcem P se premaknemo za eno v desno. V naslednjem koraku primerjamo znak na mestu T[10] z znakom na mestu P[0].

T	a	b	a	c	a	a	b	a	c	c	a	b	a	c	a	b	a	a	b	b
P											a	b	a	c	a	b				
j											0	1	2	3	4	5				
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Korak 4a do 4f:
Znak na mestu P[1] je enak znaku na mestu T[11], znak na mestu P[2] je enak znaku na mestu T[12], znak na mestu P[3] je enak znaku na mestu T[13], znak na mestu P[4] je enak znaku na mestu T[14]. Ker smo prišli do konca vzorca P, smo našli podniz v besedilu T, ki je znak po znaku enak iskanemu vzorcu P. Algoritem bi v tem primeru vrnil število 10, ki označuje začetni indeks ujemanja vzorca P v besedilu T.

Če primerjamo iskanje z algoritmom Groba Sila, vidimo, da je do razlike prišlo pri koraku 0, ko smo se pri Algoritmu Knuth-Morris-Pratt lahko premaknili za 4 v desno, pri Grobi Sile pa bi se le za 1. Poglejmo, zakaj smo se lahko s pomočjo algoritma Knuth-Morris-Pratt premaknili za 4 mesta v desno.
Označimo mesto neujemanja z rdečim pravokotnikom.

T	a	b	a	c	a	a	b	a	c	c	a	b	a	c	a	b	a	a	b	b
P	a	b	a	c	a	b														
j	0	1	2	3	4	5														
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

S celotnim vzorcem P se premaknemo za eno mesto v desno od mesta neujemanja.

T	a	b	a	c	a	a	b	a	c	c	a	b	a	c	a	b	a	a	b	b
P							a	b	a	c	a	b								
j							0	1	2	3	4	5								
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Sedaj pa pogledamo vrednost funkcije F(5), ki nam bo povedala, za koliko mest preveč v desno smo se premaknili. F(5) = 2 nam pove, da se je podniz "ab" že pojavil v vzorcu P. Zato se moremo premakniti za dve mesti nazaj. Tako ne bomo izpustili morebitnega ujemanja.

T	a	b	a	c	a	a	b	a	c	c	a	b	a	c	a	b	a	a	b	b
P					a	b	a	c	a	b										
j					0	1	2	3	4	5										
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Prav tako, je do razlike prišlo, na koraku 2, kjer smo se premaknili za 5 mest v desno, pri Grobi Sili pa bi se premaknili le za en znak v desno. Poglejmo, zakaj smo se lahko premaknili za 5 mest v desno. Označimo mesto neujemanja.

T	a	b	a	c	a	a	b	a	c	c	a	b	a	c	a	b	a	a	b	b
P						a	b	a	c	a	b									
j						0	1	2	3	4	5									
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Sedaj se s celotnim vzorcem P premaknemo tako, da poravnamo P[0] s T[10].

T	a	b	a	c	a	a	b	a	c	c	a	b	a	c	a	b	a	a	b	b
P											a	b	a	c	a	b				
j											0	1	2	3	4	5				
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

V funkcijo F pogledamo za koliko smo se zmotili. F(4) = 1. To pomeni, da se je podniz "a" pojavil v vzorcu P, od mesta P[0] do P[4], vsaj dvakrat. Torej smo se se s celotnim vzorcem P premaknili za eno mesto preveč v desno. Popravimo napako.

T	a	b	a	c	a	a	b	a	c	c	a	b	a	c	a	b	a	a	b	b
P										a	b	a	c	a	b					
j										0	1	2	3	4	5					
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Algoritem Groba Sila pa si ne zapomne nobene informacije o vzorcu P. Tako ne more premakniti vzorca P več kot za eno mesto.

1.3.4 Algoritem

Algoritem bo sestavljen iz dveh metod. Prva je `funkcijaNeuspeha(String p)`, ki nam izračuna fukcijo F in samega algoritma, `algoritemKnuthMorrisPratt(String t, String p)`.

```
import java.util.*;

public class TekstovniAlgoritmi
{
  public static int algoritemKnuthMorrisPratt(String t, String p)
  {
    //implementacija algoritma Knuth Morris Pratt
  }

  private static int[] funkcijaNeuspeha(String p)
  {
    //implementacija funkcije F(x)
  }
}
```

S pomočjo metode `funkcijaNeuspeha(String p)` bomo zgradili tabelo, ki predstavlja vrednosti funkcije F(k). Vhodni parameter je vzorec, ki ga iščemo pri algoritmu, in sicer `p` tipa `String`. Metoda vrne

tabelo celih števil tipa `int`, kjer vrednost elementa z indeksom k predstavlja vrednosti funkcije neuspeha F(k) za dani vzorec P.

```
private static int[] funkcijaNeuspeha(String p)
{
  //naredimo tabelo za F(x), ki je enaka dolžini niza p
  int[] fx = new int[p.length()];
  fx[0] = 0; // F(0) = 0 -> upoštevamo P1
  int m = p.length(); //dolžina niza p
  // z indeksom j se bomo premikali po nizu p od mesta 0 pa do m-1
  int j = 0;
  // z indeksom k se bomo premikali po nizu p od mesta 1 pa do m-1
  int k = 1;
  while(k < m)
  {
    //preverimo, če sta j-ti in k-ti znak v nizu p enaka
    if(p.charAt(j) == p.charAt(k))
    {
      //znaka sta enaka in povečamo vrednost F(k) za j+1
      //-> upoštevamo P2
      fx[k] = j+1;
      k = k+1;
      j = j+1;
    }
    else if(j > 0)
    {
      //če znaka nista enaka in je j > 0, nastavimo vrednost j
      //-> upoštevamo P4
      j = fx[j-1];
    }
    else
    {
      //če nobeden od pogojev ni izpolnjen, to pomeni, da ujemanja
      //ni in da je j < 0, nastavimo F(k) = 0 -> upoštevamo P3
      fx[k] = 0;
      k++;
    }
  }
  return fx; //vrnemo tabelo vrednosti F(k)
}
```

To metodo bomo uporabili v metodi `algoritemKnuthMorrisPratt(String t, String p)`. Slednja kot vhodna parametra dobi niza `p` in `t` tipa `String`. Kot izhodni parameter bo vrnila celoštevilsko spremenljivko, ki bo označevala začetno mesto ujemanja niza `p` v nizu `t`. Če niz `t` ne vsebuje podniza, ki se znak za znakom ujema z vzorcem `p`, bo metoda vrnila `-1`.
Algoritem je sledeč:

> Primerjamo znak na mestu `t[i]` z znakom na mestu `p[j]`. Če se znaka ujemata nadaljujemo s primerjanji, sicer pogledamo ali je spremenljivka `j > 0`. Če je `j > 0`, pomeni, da smo ujemanje vzorca `p` v besedilu `t` našli od `0` do `j - 1`. Pogledamo v tabelo `F` in odčitamo vrednost `F(j - 1)`. Če je vrednost `F(j - 1) = 0`, pomeni, da se s celotnim vzorcem `p` premaknemo tako, da bomo na naslednjem koraku primerjali vrednost na mestu `t[i]` s `p[0]`. Če je `F(j - 1) > 0`, potem se s celotnim vzorcem `p` premaknemo tako, da bomo na naslednjem koraku primerjali znak na mestu `t[i + 1 - F(j - 1)]`. Če je `j <= 0` in se znaka ne ujemata, se s celotnim vzorcem `p` premaknemo tako, da bomo na naslednjem koraku primerjali znak na mestu `t[i + 1]` z znakom `p[0]`.

```
public static int algoritemKnuthMorrisPratt(String t, String p)
{
  int n = t.length(); //dolžina besedila T
  int m = p.length(); //dolžina vzorca P

  //pokličemo metodo, ki nam zgradi F(x)
  int[] fx = funkcijaNeuspeha(p);
  int j = 0; //indeks, ki bo tekel po vzorcu P
  int i = 0; //indeks, ki bo tekel po besedilu T

  //dokler ne pridemo do konca besedila T ponavljamo naslednje korake
  while(i < n)
  {
    //če se j-ti znak v vzorcu P ujema z i-tim znakom v besedilu T
    if(p.charAt(j) == t.charAt(i))
    {
      //če pridemo do konca vzorca p (j == m-1) smo našli ujemanje
      //in vrnemo začetni indeks ujemanja
      if(j == m-1) return i-m+1;
      else //sicer povečamo indeksa za ena
      {
        i = i+1;
        j = j+1;
      }
    }
    //če ujemanj ni in je j > 0, pogledamo v tabelo F(x) (na mestu j-1
    //smo našli zadnje ujemanje)
    else if(j > 0) j = fx[j-1];
    //sicer povečamo indeks, ki teče po besedilu T, za ena
    else i = i+1;
   }
   return -1; //ujemanja nismo našli, vrnemo -1
}
```

1.3.5 Časovna zahtevnost

Oglejmo si časovno zahtevnost algoritma.
Velikost problema, ki ga obravnavamo, je odvisna od dolžine obeh nizov, ki sta vhodna parametra. Kot vhodna parametra algoritem sprejme besedilo T in vzorec P. Dolžini niza T, ki jo označimo z `n` in niza P, ki jo označimo z `m`, predstavljata velikost problema. Za algoritem Knuth-Morris-Pratt, ki ga označimo s `c`, določimo časovno zahtevnost problema $T_c(n, m)$.

Algoritem kliče metodo `funkcijaNeuspeha(p)`, v kateri se `m`-krat sprehodimo po vzorcu `p`. Koliko korakov naredimo na vsakem sprehodu, je odvisno od tega, kako smo napredovali v vzorcu `p`. To je odvisno od vrednosti `F(k)`, ki smo jih do tega koraka že izračunali. V vsakem primeru pa je število korakov vedno konstantno. Zato metoda porabi `O(m)` časa za izvajanje. Algoritem je sestavljen še iz zanke `while`, v kateri se s pomočjo indeksa `i`, ki teče od `0` do vključno `n - 1`, sprehodimo po besedilu T. Dokler je še kaj znakov v besedilu T, primerjamo, ali vzorec P vsebuje trenutni znak iz besedila T. Tu se `n`-krat sprehodimo po besedilu T in na vsakem koraku opravimo natanko eno primerjanje, in sicer pogledamo ali vzorec P vsebuje trenutni znak iz besedila T. Časovna zahtevnost algoritma Knuth-Morris-Pratt je seštevek primerjanj, ki jih izvedemo, ko gradimo funkcijo neuspeha in primerjanj v zanki `while`. To je `O(m) + O(n) = O(n + m)`.

Oglejmo si kakšna je časovna zahtevnost algoritma v najboljšem, pričakovanem in najslabšem primeru.

Najboljši primer

Če zanemarimo očitno nesmiselen primer, ko je dolžina vzorca P daljša od dolžine besedila T, se v najboljšem primeru vzorec P nahaja na začetku besedila T. Takrat dolžina besedila T ne vpliva na časovno zahtevnost. Pri gradnji `F(k)` porabimo `O(m)` časa za izvajanje. Za vsak znak v vzorcu P opravimo eno primerjavo, torej je število primerjanj enako `m`.

$$T_c(n, m) = O(m) + O(m) = O(2 * m) = O(m)$$

Najslabši primer
V najslabšem primeru vzorca v besedilu sploh ni. Pri tem bo najslabši primer nastopil takrat, ko bomo pri preverjanju v notranji podzanki vedno prišli prav do konca. Premislek nam pokaže, da bo tak najslabši par denimo "xxxxxxxxz" in "xxy". V tem primeru se izvede maksimalno število primerjanj.

$$T_c(n, m) = O(n + m)$$

Najslabši primer nastopi tudi takrat, ko je vzorec sestavljen skoraj ali v celoti iz samih različnih znakov. To pomeni, da je pri grajenju funkcije `F(k)` verjetnost neujemanja velika. Pri algoritmu Groba Sila in algoritmu Boyer-Moore smo kot primer delovanja algoritma uporabili besedilo T = "Algoritem za iskanje" in vzorec P = "iskan" oziroma P = "iskanje". Vse črke iz vzorca P so različne, niti ena se ne ponovi dvakrat. Pri grajenju `F(k)` bi opazili, da so vse njene vrednosti enake nič. To pa pomeni, da bi se z vzorcem P po danem besedilu T premikali vedno samo za en znak naprej .

Pričakovana časovna zahtevnost
Ker je analiza pričakovane časovne zahtevnosti zapletena, povejmo le, da je enaka `O(n + m)`.

2 Stiskanje teksta (Text Compression)

Stiskanje teksta je tehnika, ki zmanjša velikost datoteke ali besedila, ne da bi vplivala na informacije (besede), ki so vsebovane v datoteki oziroma besedilu. Ena od teh tehnik stiskanja teksta je Huffmanovo kodiranje, ki je predstavljeno v tem poglavju.

Vzemimo tekstovno datoteko, katere znaki so zapisani v kodi ASCII. Naj datoteka vsebuje besedilo, sestavljeno iz 1500 znakov. To je približno število znakov, ki gredo na stran A4. Ker je besedilo zapisano v kodi ASCII, vsak znak v besedilu za shranjevanje potrebuje 8 bitov prostora. Tako je na primer znak 'A', ki ima v desetiškem zapisu kodo 65, shranjen kot 01000001. Tako bi za celotno besedilo porabili 12.000 bitov shranjevalnega prostora (1500 znakov pomnožimo z osmimi biti).

S pomočjo metode stiskanja teksta porabimo veliko manj shranjevalnega prostora, nekje od 20% pa do 90 %, odvisno od porazdelitve pogostosti znakov v besedilu. Če zgoraj omenjeno besedilo stisnemo, je elektronski prenos besedila bistveno hitrejši.

2.1.1 Huffmanovo kodiranje

Pri problemu stiskanja teksta imamo dano besedilo X. Besedilo X je sestavljeno iz znakov neke abecede A. X želimo učinkovito zakodirati v binarni zapis, ki ga bomo predstavili z nizom Y. Torej bo niz Y sestavljen iz znakov 0 in 1.

Standardne sheme za kodiranje, kot na primer ASCII in Unicode, uporabljajo za kodiranje znakov fiksno dolžino binarnega zapisa (ASCII uporablja 8 bitov, Unicode pa 16). Huffmanova koda za razliko od omenjenih za kodiranje znakov uporablja spremenljivo dolžino binarnega zapisa. Huffmanova koda porabi manj prostora kot kodiranje s fiksno dolžino, ker znake, ki se v besedilu uporabljajo pogosteje, zakodiramo s krajšim zapisom, znake, ki se redkeje ponavljajo, pa z daljšim zapisom. Temu rečemo kodiranje s spremenljivo dolžino.

Poglejmo enostaven zgled. Naj bo dan niz "abba". Na običajni način znak 'a' zakodiramo z 01100000, 'b' pa z 01100001. Niz "abba" je torej zakodiran kot 01100000011000010110000101100000, torej z 32 biti. Lahko pa bi rekli, da bomo 'a' kodirali z 1 in 'b' z 0, ter tako "abba" kodirali kot 1001 in porabili le 4 bite!
Seveda se stvar zaplete, če v nizu nastopa več znakov. Denimo da je X = "abbaaaccb". Takrat se moramo seveda odločiti, kateremu znaku bi pripadal krajši zapis, pa tudi, kako sestaviti zapis, da bomo potem zapis lahko odkodirali nazaj. Če bi rekli kar, da npr. 'a' kodiramo z 0, 'b' z 1, 'c' pa z 01, kako bomo vedeli, kaj pomeni 1001. Je to niz "baab" ali pa je to niz "bac"?

Pri kodiranju s spremenljivo dolžino imamo torej lahko težave z odkodiranjem. Temu se lahko izognemo tako, da sestavimo kodirno drevo.
Kodirno drevo je dvojiško drevo, ki ima toliko listov, kot imamo znakov abecede (za zadnji navedeni zgled bi torej potrebovali drevo s tremi listi). V liste razporedimo znake. Kodiranje znakov izvedemo tako, da z 0 označimo povezavo iz notranjega vozlišča do levega sina, z 1 pa do desnega sina. Kodni zapis izbranega znaka iz niza X je zaporedje oznak na povezavah, ki vodijo od korena do lista, v katerem hranimo znak.
Če drevo in kode znakov sestavimo po omenjenem postopku, nimamo problemov s predponami. Vsak znak je shranjen v svojem listu drevesa. Do vsakega lista v drevesu pridemo natanko po eni poti. Iz tega sledi, da nobena kodna beseda ne more biti predpona druge kodne besede. Če bi hranili znake tudi v notranjih vozliščih, potem bi imeli probleme s predponami.

Pri Huffmanovi kodi tudi uporabljamo kodirno drevo. Ko ga sestavljamo, smo pozorni na pogostost, s katero znaki nastopajo v besedilu. V listih drevesa T hranimo znake iz niza X in njihove relativne frekvence v besedilu. V notranjih vozliščih drevesa je zapisana vsota frekvenc obeh sinov. Takrat nam vrednost v notranjih vozliščih pove, kolikšna je pogostost pojavitve skupine znakov, ki jih hranimo v listih.

Koraki Huffmanovega algoritma

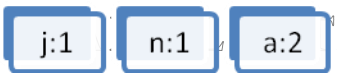
- Izračunamo pogostost znakov v besedilu X.
- Zgradimo drevo.
- S pomočjo drevesa določimo kode.
- Zakodiramo besedilo.
- Odkodiramo besedilo.

Oglejmo si zgoraj napisane korake na enostavnem primeru. Naj bo niz X = "anja". Abeceda A je sestavljena iz znakov, ki nastopajo v nizu X, torej A = {'a', 'j', 'n'}.

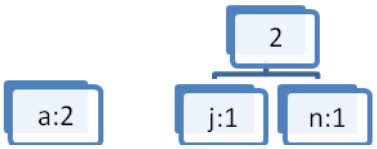
Izračunamo pogostost znakov v besedilu X.
Zanima nas, koliko krat se znak 'a' pojavi v nizu X. To označimo z f('a') in ji rečemo frekvenca znaka 'a'. Naredimo tako za vse znake abecede in dobimo tabelo:

znak	f(znak)
a	2
j	1
n	1

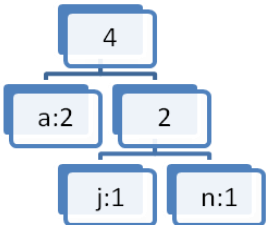
Zgradimo drevo frekvenc za dani niz X.
Najprej razporedimo znake po pogostosti ponavljanja, in sicer od znakov, ki se najmanjkrat ponovijo, do znakov, ki se največkrat ponovijo. Znaki, ki se ponovijo enako pogosto, so načeloma razporejeni poljubno. Zato imamo v splošnem lahko več različnih dreves, ki jih sestavimo iz besedila X. Torej lahko isto besedilo kodiramo različno (posamezni znaki imajo v posameznem kodiranju različne kode). Vsa pa bodo enako uspešna (kratka).
Vsak znak s pripadajočo frekvenco predstavlja Huffmanovo drevo. Torej bomo za naš primer na začetku imeli tri Huffmanova drevesa. Na vsakem koraku bomo poiskali Huffmanovi drevesi z najmanjšima frekvencama in ju bomo združili v večje Huffmanovo drevo.



Z gradnjo drevesa začnemo tako, da vzamemo Huffmanovi drevesi, ki imata v korenih podatka z najmanjšo vrednostjo (torej se znaki v njih najmanjkrat ponovijo) in ju združimo v Huffmanovo drevo. V korenu združenega drevesa je vsota njunih frekvenc. Dobljeno drevo predstavlja oba znaka (v našem primeru 'j' in 'n'), podatek v korenu (2) pa kako pogosto ta dva znaka nastopata v besedilu.

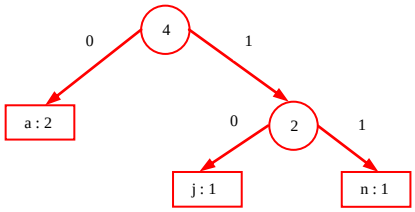


Ponovno vzamemo dve Huffmanovi drevesi z najmanjšima frekvencama (v našem zgledu sicer nimamo izbire, saj sta na voljo le še dve drevesi) in ju združimo v večje Huffmanovo drevo. Koren drevesa je vsota njunih frekvenc.



Z gradnjo drevesa zaključimo, ko imamo samo eno drevo.

S pomočjo zgrajenega drevesa določimo kode znakov.
Z 0 označimo povezavo iz notranjega vozlišča do levega sina, z 1 pa do desnega sina.



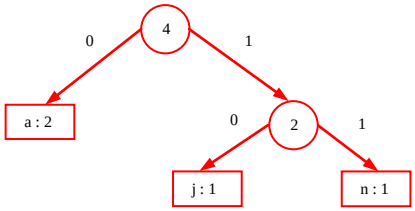
Kode znakov preberemo od korena navzdol. Kodni zapis znaka 'j' preberemo tako, da začnemo v korenu in se premikamo proti listu drevesa, kjer hranimo znak. Zato je kodni zapis znaka 'j' enak 10. Napišimo kode znakov v tabelo.

znak	koda znaka
a	0
j	10
n	11

Zakodiramo besedilo.
Združimo vse kode znakov in dobimo zakodiran niz Y, ki pripada nizu X.
Y = "011100".

Odkodiramo besedilo.
Pri Huffmanovem algoritmu je poleg zakodiranega besedila potrebno hraniti tudi kode znakov, da znamo besedilo tudi odkodirati (ali pa kar kodirno drevo). Pri fiksnih kodah (ASCII, Unicode, …) kod znakov ni potrebno hraniti, saj se kode znakov ne spreminjajo in so vedno enake.

Oglejmo si, kako poteka odkodiranje za primer zakodiranega besedila Y = "011100". Za odkodiranje potrebujemo še drevo

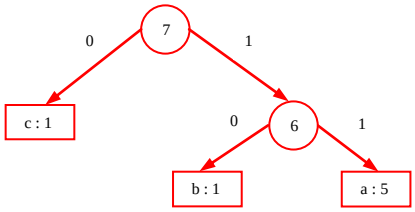


Korak 0:
Odkodiranje začnemo kar pri prvem znaku iz niza Y, znakom '0'. Sledimo povezavi iz korena drevesa do levega sina. Ker je levi sin enak listu, je prvi odkodirani znak enak 'a'. Pobrišemo znak '0' iz niza Y. Niz Y = "11100".
Korak 1:
Naslednji znak v nizu Y je '1'. Sledimo povezavi iz korena do desnega sina. Ker še nismo prišli do lista, odreženo iz niza Y znak '1' in vzamemo naslednji znak, znak '1'. Sledimo povezavi iz notranjega vozlišča z vrednostjo 2 do desnega sina. Prišli smo do lista, zato je naslednji odkodirani znak enak 'n'. Odrežemo znak '1' iz niza Y in Y = "100".
Korak 3:
Prvi znak iz odrezanega niza Y je '1'. Sledimo povezavi od korena do desnega sina. Prišli smo do notranjega vozlišča z vrednostjo 2. Zato pogledamo naslednji znak iz Y, znak '0'. Iz notranjega vozlišča sledimo povezavi do levega sina. Prišli smo do lista. Naslednji odkodiran znak je znak 'j'. Odrezani niz Y = "0".
Korak 4:
V nizu Y nam je ostal le še znak '0', ki nas vodi od korena drevesa do levega sina, ki je list drevesa. Zadnji odkodirani znak je enak 'a'.

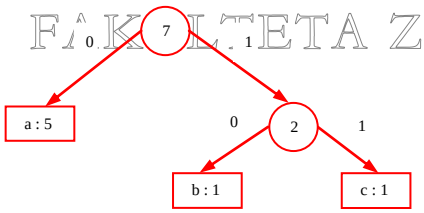
Odkodirani niz je enak nizu X = "anja".

Pri grajenju drevesa smo na vsakem koraku vzeli dve Huffmanovi drevesi z najmanjšima frekvencama in ju združili v večje Huffmanovo drevo. S tem smo dosegli, da imajo na primer znaki, ki se ponovijo le enkrat v besedilu X daljši zapis, kot znaki, ki se v besedilu X ponovijo denimo 5 krat. Tisti znaki, ki so najbolj pogosti, bodo imeli torej kratko kodo. S tem bomo za kodiranje celotnega besedila porabili manj prostora.

Primer:
Naj bo X = "aaaaabc".
Denimo, da bi zgradili drevo tako, da nismo pozorni na pogostost znakov v besedilu X.

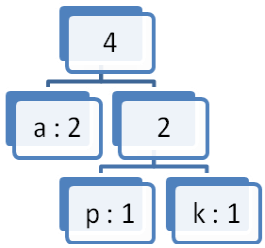
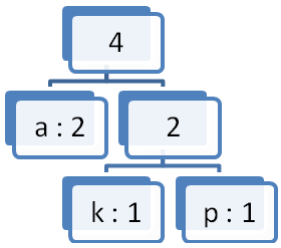
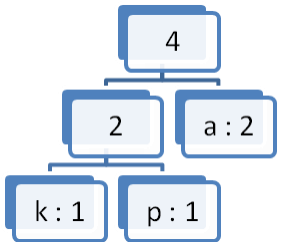
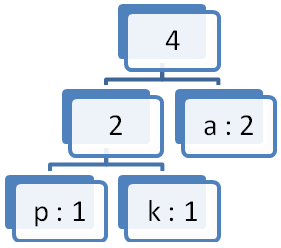


Tako bo za drevo na sliki koda znaka 'a' enaka 11, 'b' 10 in 'c' 0. Zakodiran niz Y, ki pripada nizu X, je enak "1111111111100".
Sedaj upoštevamo pogostost pojavitve znakov v besedilu X in sledimo prej opisanemu postopku. Dobimo drevo



Koda znaka 'a' bo enaka 0, 'b' 10 in 'c' 11. Niz Y = "000001011".
V prvem primeru zakodiramo besedilo X s 13-timi biti, v drugem primeru pa z 9-timi biti. Drugi pristop je učinkovitejši, saj porabimo manj prostora.

Omenili smo, da lahko za besedilo X zgradimo več dreves, ki bodo vsa dala najkrajše kodiranje. Pa si poglejmo to na primeru.
Naj bo X = "kapa". Ustrezna kodirna drevesa in kode znakov iz besedila X so:

drevo 1		**znak** / **koda znaka** a / 0 p / 10 k / 11	Y = 110100
drevo 2		**znak** / **koda znaka** a / 0 p / 11 k / 10	Y = 100110
drevo 3		**znak** / **koda znaka** a / 1 p / 01 k / 00	Y = 001011
drevo 4		**znak** / **koda znaka** a / 1 p / 00 k / 01	Y = 011001

Kot vidimo, smo v vseh primerih za kodiranje besedila Y porabili 6 bitov.

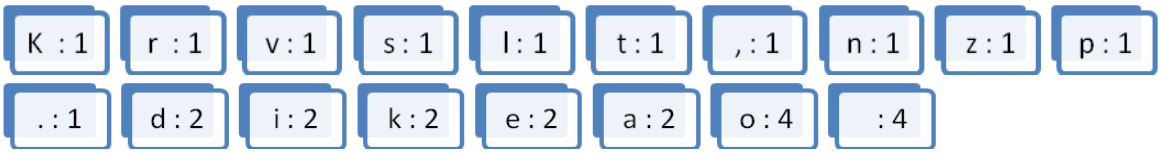
2.1.2 Primer

Oglejmo si sedaj malo daljši primer. Dan imamo niz X = "Kdor visoko leta, nizko pade.". Znaki iz niza X pripadajo znakom iz abecede A = { , ',', ., a, d, e, i, k, l, n, o, p, r, s, t, v, z, K} (prvi znak je presledek). Dolžina abecede A je d = 18.
Najprej zgradimo tabelo frekvenc za vsak znak iz niza X.

znak	f(znak)
	4
,	1
.	1
a	2
d	2
e	2
i	2
k	2
l	1
n	1
o	4
p	1
r	1
s	1
t	1
v	1
z	1
K	1

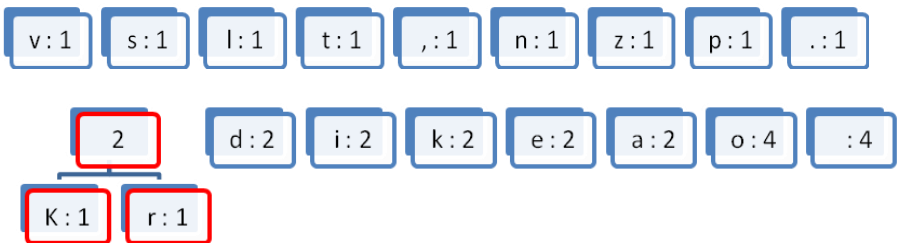
Sestavimo Huffmanovo drevo:

Korak 0:
Znake in njihove frekvence si predstavljamo kot Huffmanova drevesa. Pogostost ponavljanja Huffmanovega drevesa z enim samim vozliščem pomeni pogostost pojavitve znaka v nizu X. Razporedimo Huffmanova drevesa po pogostosti ponavljanja, in sicer od dreves, ki se najmanjkrat ponovijo, do dreves, ki se največkrat ponovijo. Drevesa, ki se ponovijo enako pogosto, so načeloma razporejena poljubno.

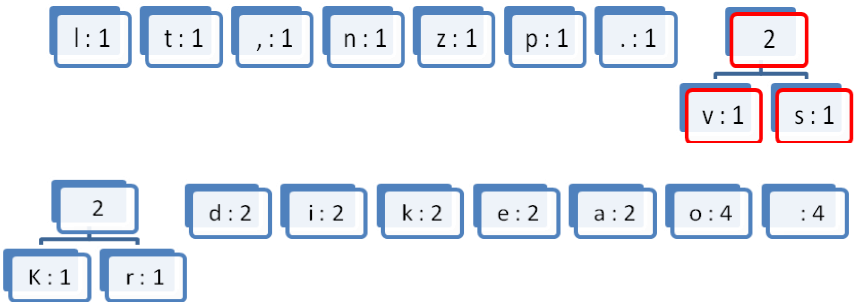


Korak 1:
Drevo začnemo graditi tako, da vzamemo dve Huffmanovi drevesi, ki se v nizu T ponovita najmanjkrat. Ker imamo na voljo kar 11 dreves s pogostostjo 1, imamo res veliko možnih izbir. Tako lahko vzamemo drevesi, ki vsebujeta znaka 'K' in 'r' ali 'K' in 's' ali 't' in ',' ali ...

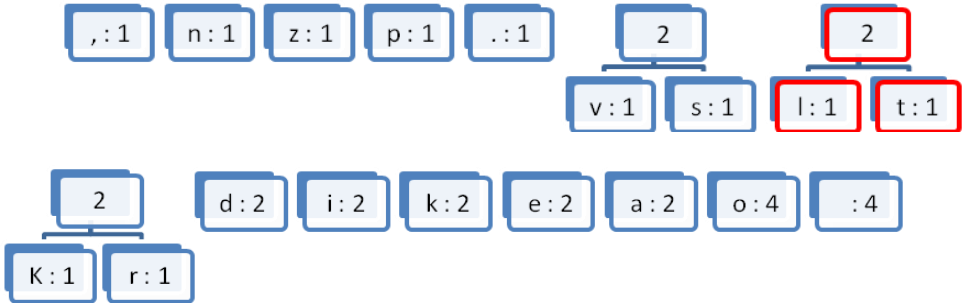
Denimo, da vzamemo kar drevesi, ki vsebujeta znaka 'K' in 'r'. Združimo drevesi v skupno drevo. V korenu združenega drevesa shranimo podatek, ki je seštevek ponovitve prvega in drugega znaka, 2. S tem smo povedali, da je skupna pogostost pojavitve znaka 'K' in znaka 'r' enaka 2.
Algoritem bo novo Huffmanovo drevo vstavil v vrsto nekam med tista drevesa, ki imajo frekvenco 2. Denimo, da bo novo drevo vstavil na začetek vrste, kjer se pojavijo drevesa s pogostostjo 2.



Korak 2:
Nadaljujemo z združevanjem. Spet vzamemo Huffmanovi drevesi, ki se v nizu T ponovita najmanjkrat. Ponovno imamo več možnosti. Lahko vzamemo drevesi, ki hranita znaka 'v' in 's' ali 'p' in '.' ali … Vzemimo drevesi, ki hranita znaka 'v' in 's' in ju združimo v drevo. Seštevek pojavitve drevesa, ki hrani znak 'v' in drevesa, ki hrani znak 's' je enak 2. Seštevek shranimo v koren novega drevesa.

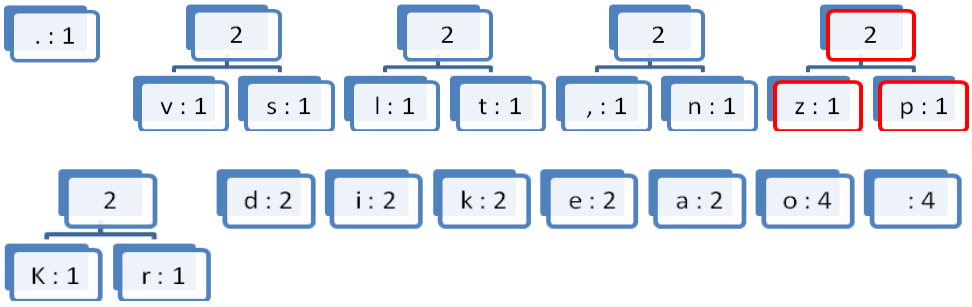


Korak 3:
Nadaljujemo z združevanjem. Drevesi, ki predstavljata znaka, ki se najmanjkrat ponovita v nizu T, sta 'l' in 't'. Združimo ju v skupno drevo. Seštevek pojavitve drevesa, ki vsebuje znak 'l' in drevesa, ki vsebuje znak 't' je enak 2. Seštevek shranimo v koren združenega drevesa.



Korak 4:
V drevo združimo drevesi z znakoma ',' in 'n'.

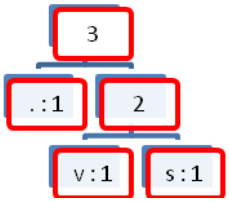
Korak 5:
V drevo združimo drevesi z znakoma 'z' in 'p'. Vrsta je po tem koraku enaka:



Vidimo, da v našem besedilu nastopa znak '.' s pogostostjo ena, skupina znakov 'v' in 's' s pogostostjo 2. Tudi pari znakov 'l' in 't', ',' in 'n', ... nastopajo s pogostostjo 2 (to pomeni, da se v besedilu s pogostostjo 2 pojavi npr. ali znak 'l' ali znak 't'). S pogostostjo 2 se pojavijo tudi znaki 'd', 'i', ..., znaka 'o' in ' ' pa se v besedilu pojavita s pogostostjo 4.

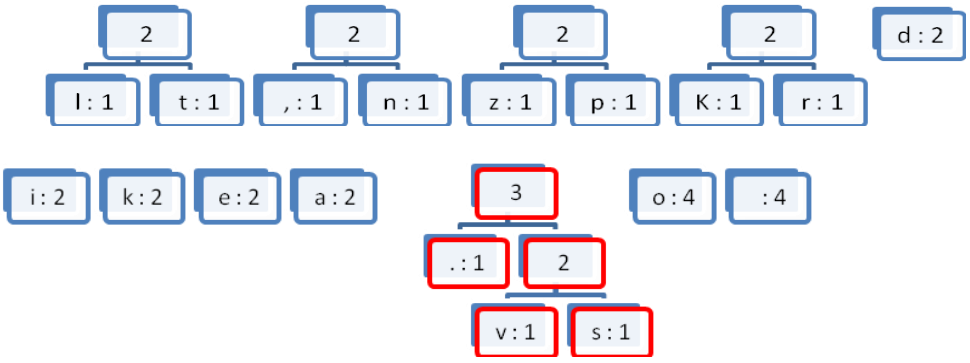
Korak 6:
Sedaj nimamo več na voljo dveh Huffmanovih dreves s pogostostjo 1, saj s pogostostjo 1 nastopa le še Huffmanovo drevo, ki hrani znak '.'. Zato v skupno drevo združimo drevesi, ki vsebujeta znak '.' ter eno

izmed dreves, ki ima v korenu število 2. To je bodisi eno izmed dreves, ki hrani posamezni znak s pogostostjo 2 bodisi drevo, ki predstavlja skupino znakov (v našem primeru dveh), ki se pojavljata s skupno pogostostjo 2. Vzemimo kar prvo drevo v vrsti s pogostostjo 2. Torej drevo, ki v listih hrani znaka 'v' in 's'. To drevo nam predstavlja, kako pogosto se v besedilu pojavita bodisi 's' bodisi 'v'. Dobimo



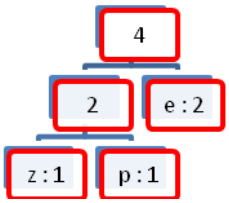
To je Huffmanovo drevo za znake '.', 'v' in 's'. Vrednost v korenu (3) nam pove, da se ti znaki v besedilu T pojavijo s skupno pogostostjo 3.

Algoritem vstavi novo drevo v vrsto, takoj po zadnjem Huffmanovem drevesu s pogostostjo 2.

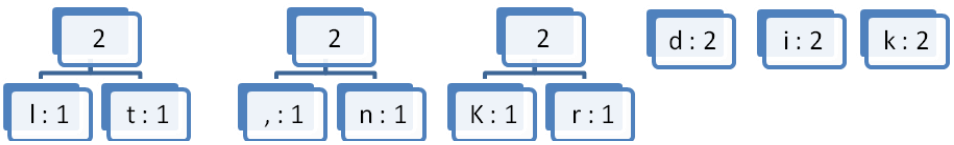


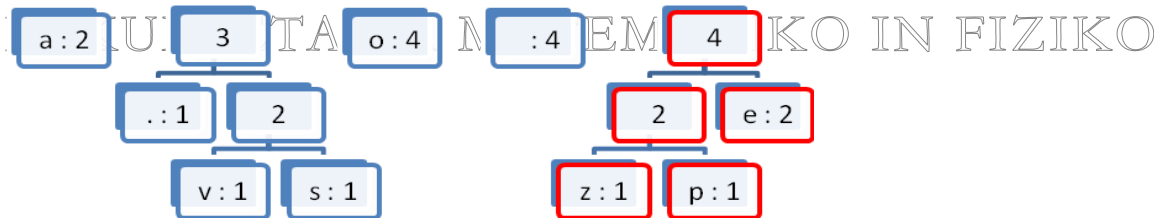
Korak 7:
Na tem koraku lahko v drevo združimo na primer drevo, ki ima v korenu vrednost 2 in v listih hrani znaka 'z' in 'p' in drevo, ki ima v korenu vrednost 2 in hrani znaka 'K' in 'r' ali drevesi, ki hranita znaka 'i' in 'e' ali drevo, ki hrani znaka ',' in 'n' in drevo z znakom 'a' ali … Vidimo, da je možnosti precej. V vsakem primeru pa bomo dobili drevo, ki bo v korenu imelo vrednost 4. Predstavljalo nam bo torej skupino znakov, ki se v besedilu pojavljajo s (skupno) pogostostjo 4.

Recimo, da vzamemo drevo, ki v listih hrani znaka 'z' in 'p' in drevo, ki hrani znak 'e' in ju združimo v skupno drevo. V koren zapišemo vrednost 4.



Algoritem bo narejeno drevo vstavil v vrsto, kjer so drevesa s pogostostjo 4. Ker je vseeno na katero mesto med drevesi s pogostostjo 4, denimo, da ga bo vstavil na konec vrste.

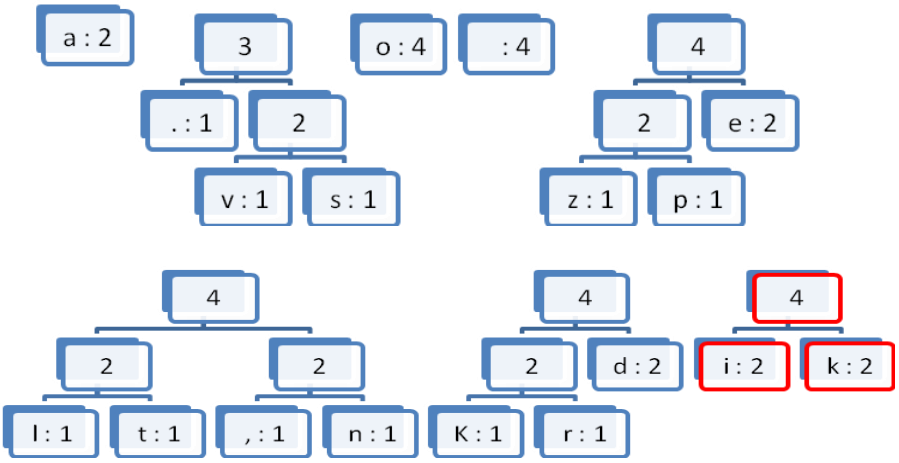




Korak 8:
Vzamemo drevesi, ki imata v korenu vrednost 2 in v listih hranita znake 'l', 't', ',', n in ju združimo v skupno drevo. Vrednost v korenu je 4.

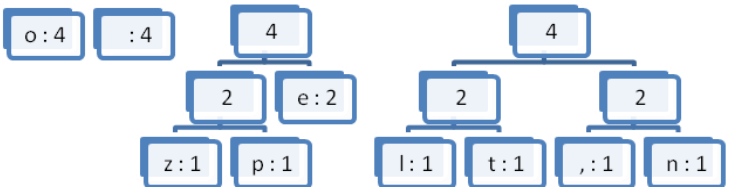
Korak 9:
Ponovno vzamemo drevo, ki v korenu hrani vrednost 2, v listih pa znaka 'K', 'r' in drevo, ki vsebuje znak d. Združimo ju v skupno drevo, vrednost v korenu je 4.

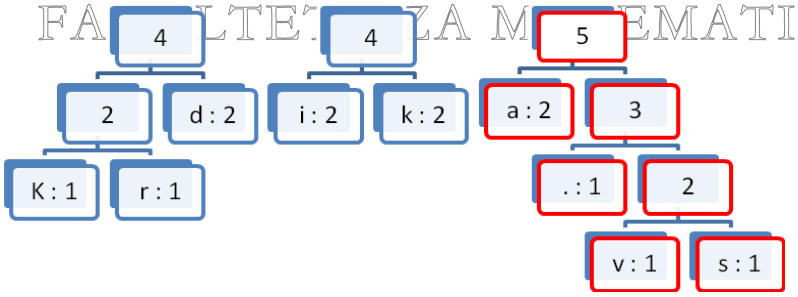
Korak 10:
V drevo združimo drevesi, ki hranita znaka 'i' in 'k'. Vrednost združenega drevesa v korenu je 4. Vrsta je po tem koraku enaka.



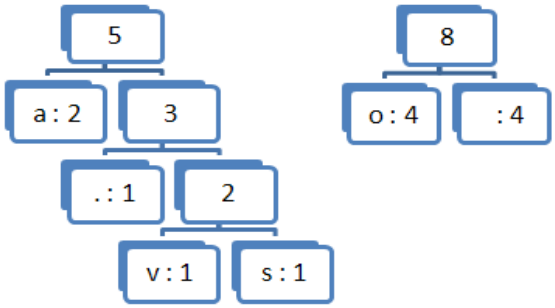
Iz vrste lahko preberemo, da v danem besedilu X nastopa znak 'a' s pogostostjo 2, skupina znakov '.', 'v', 's' s pogostostjo 3. S pogostostjo 4 se pojavita znaka 'o' in ' '. Tudi skupine znakov {'l', 't', ',', 'n'}, {'z', 'p', 'e'}, {'K', 'r', 'd'} in {'i', 'k'} se pojavijo s pogostostjo 4. To pomeni, da se v besedilu s pogostostjo 4 pojavi znak 't' ali znak 'k' ali znak 'p', …

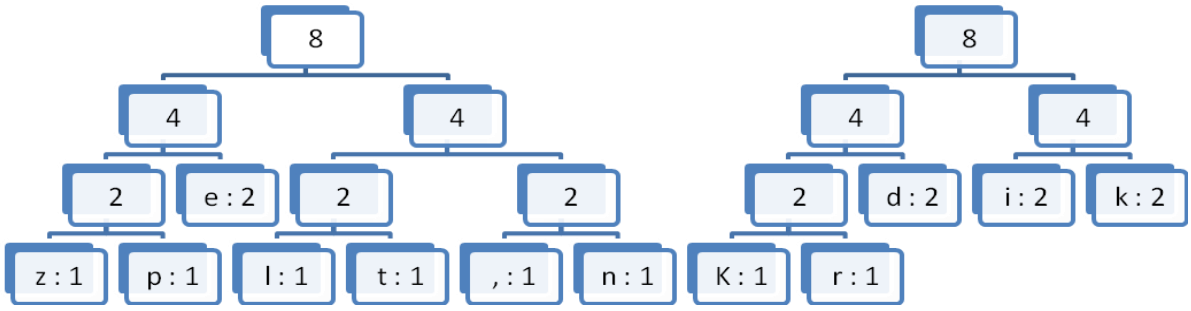
Korak 11:
Na tem koraku imamo samo eno možnost. V drevo lahko združimo le drevo, ki hrani znak 'a' in drevo, ki ima v korenu vrednost 3.





Na **korakih 12, 13, 14** bomo v skupna drevesa združevali drevesa, ki imajo v korenu vrednost 4. Po **koraku 14** pridemo do naslednjega stanja

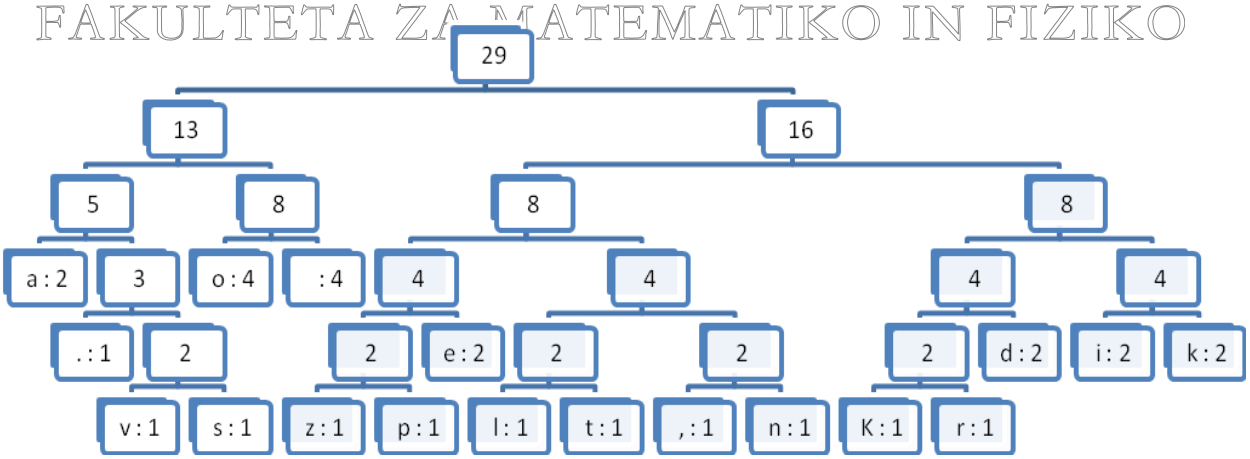




Korak 15:
V skupno drevo združimo drevo, ki ima v korenu vrednost 5 in eno imed dreves, ki imajo v korenu vrednost 8. Recimo, da vzamemo drevo, ki v listih hrani znaka 'o' in ' '. Novo drevo bo v korenu hranilo vrednost 13.

Korak 16:
Ostali sta še dve drevesi, ki v korenu hranita vrednost 8, zato ju združimo v drevo, ki bo v korenu imelo vrednost 16.

Korak 17:
Ostali sta še dve drevesi, ki ju združimo v eno drevo. Vrednost v korenu bo 29. Dobimo naslednje drevo:



Opazimo, da je v korenu 29, kar je točno dolžina niza X. To je seveda pričakovano, saj nam vrednost v korenu pove, kako pogosto se pojavi skupina znakov (tukaj so to vsi znaki) v nizu X.

Sedaj, ko imamo zgrajeno Huffmanovo drevo, lahko povezavam od vozlišča do sina dodelimo bite. Povezava, ki vodi od vozlišča do levega sina, dobi težo 0, od vozlišča do desnega sina pa 1. Lahko bi se odločili obratno, a to na samo kvaliteto kodiranja ne bi vplivalo.

Iz zgrajenega Huffmanovega drevesa za vsak znak preberemo njegovo kodo. Koda znaka je zaporedje bitov 0 in 1, ki jih za vsak znak posebej preberemo iz Huffmanovega drevesa. Prebiramo jih od korena do lista, v katerem hranimo iskani znak.

znak	**koda znaka**
	011
,	10110
.	0010
a	000
d	1101
e	1001
i	1110
k	11000
l	10100
n	10111
o	010
p	10001
r	11001
s	00111
t	10101
v	00110
z	10000
K	11000

Združimo vse kode znakov in dobimo zakodiran niz Y, ki pripada nizu X = "Kdor visoko leta, nizko pade.".
Y = 11000 1101 010 11001 011 00110 1110 00111 010 11000 010 011 10100 1001 10101 000 10110 011 10111 1110 10000 11000 010 011 10001 000 1101 1001 0010.
Če seštejemo vse 0 in 1, ugotovimo, da smo niz X zakodirali z 118 biti. Če bi isto besedilo zakodirali na običajen način, recimo s kodo ASCII, bi porabili 232 bitov (= 8 * dolžina niza). Torej bi z običajnim kodiranjem porabili skoraj še enkrat več prostora kot s Huffmanovim kodiranjem.

2.1.3 Algoritem

Pri gradnji Huffmanovega drevesa uporabimo vrsto s prednostjo. Vrsta s prednostjo je definirana glede na frekvenco znaka. Drevo začnemo z gradnjo poddreves. Poddrevesa dobimo tako, da na vsakem koraku združimo poddrevesi z najmanjšima frekvencama. Podatek v notranjih vozliščih predstavlja vsoto frekvenc levega in desnega sina poddrevesa. Povezave na leve sinove označimo z 0, na desne pa z 1. Zaporedje znakov na povezavah od lista do korena predstavljajo kodo znaka v listu.

Najprej poglejmo kaj vse potrebujemo za implementacijo Huffmanovega algoritma.
Glavno vlogo v algoritmu bo predstavljala vrsta s prednostjo. V tej vrsti bomo hranili Huffmanova drevesa za ustrezno množico znakov. Pri tem bomo kot vrednost elementa definirali podatek, ki je v korenu drevesa.
Na začetku bomo vse znake in frekvence znakov, kot Huffmanova drevesa vstavili v vrsto.

```
vrsta <- znaki
```

Dokler vrsta ne bo vsebovala le enega elementa, ponavljamo naslednje:

```
levoDrevo <- vrsta.najmanjsiElement()
vrsta.odstaniNajmanjsega()
desnoDrevo <- vrsta.najmanjsiElement();
vrsta.odstaniNajmanjsega();
koren.frekvenca <- levoDrevo.koren.frekvenca +
     desnoDrevo.koren.frekvenca;
koren.znak <- pomozniZnak
drevo <- koren + levoDrevo + desnoDrevo;
```

Grajenje Huffmanovega drevesa se zaključi, ko vrsta vsebuje le en element. Takrat lahko drevesu dodelimo kode, in sicer povezavi od notranjega vozlišča do levega sina dodelimo 0, povezavi od notranjega vozlišča do desnega sina pa 1. Preberemo kode znakov in zakodiramo besedilo X. Potrebovali bomo tudi metodo, ki zna odkodirati niz Y.

Za implementacijo Huffmanovega algoritma potrebujemo več razredov, in sicer `HuffmanovoVozlisce`, `HuffmanovoDrevo`, `VerizniSeznam`, `VrstaSPrednostjo`, `HuffmanovoKodiranje` in `HuffmanovoKodiranjeUporaba`. Oglejmo si predstavitev in uporabo omenjenih razredov.

Razred `HuffmanovoVozlisce`

Huffmanovo drevo je zgrajeno iz Huffmanovih vozlišč. Huffmanovo vozlišče je lahko list ali notranje vozlišče. V listih hranimo znak in pogostost pojavitve znaka v danem nizu `X`. V notranjih vozlišč hranimo vsoto frekvenc sinov. V implementaciji algoritma bodo notranja vozlišča imela enake lastnosti kot listi.

Razred `HuffmanovoVozlisce` bo predstavljal vozlišče v Huffmanovem drevesu z naslednjimi lastnostmi:

- `frekvenca`
- `znak`

`Frekvenca` bo tipa `int` in bo za liste drevesa predstavljala pogostost pojavitve znaka v nizu `X`. Za notranja vozlišča pa vsoto frekvenc sinov. `Znak`, ki bo tipa `char`, bo za liste predstavljal znak iz abecede `A`, ki vsebuje znake iz niza `X`. Za notranja vozlišča pa bo predstavljal pomožni znak '¤'. Dejansko ga ne bomo potrebovali.

```
public class HuffmanovoVozlisce
{

  char znak; //znak iz dane abecede A
  int frekvenca; //pogostost pojavitve znaka v danem nizu X

}
```

Razred `HuffmanovoDrevo`

Vsako Huffmanovo drevo je sestavljeno iz korena, levega poddrevesa in desnega poddrevesa.

```
public class HuffmanovoDrevo{

  HuffmanovoVozlisce koren; //kazalec na koren v Huffmanovem drevesu

  //kazalec na levo poddrevo v Huffmanovem drevesu
  HuffmanovoDrevo levo;
  //kazalec na desno poddrevo v Huffmanovem drevesu
  HuffmanovoDrevo desno;

}
```

Razred `VerizniSeznam`

Predstavljal bo verižni seznam Huffmanovih dreves. V seznamu bomo vedno lahko dostopali do prvega Huffmanovega drevesa. Zato potrebujemo še kazalec na preostanek verižnega seznama.

```
public class VerizniSezman
{

  HuffmanovoVozlisce vozlisce; //Huffmanovo vozlisce
  VerizniSeznam naslednji; //kazalec na preostanek verižnega seznama

}
```

Razred `VrstaSPrednostjo`

Vrsta s prednostjo bo vsebovala verižni seznam Huffmanovih dreves. S pomočjo razreda VrstaSPrednostjo bomo hranili Huffmanova drevesa v verižnem seznamu in jih bomo urejali.

Uporabili bomo "slabo" implementacijo vrste s prednostjo. Bolj smiselno bi bilo uporabiti na primer implementacijo z minimalno kopico. Na koncu tega podpoglavja bomo razložili idejo, kako bi lahko izboljšali ta razred.

Razred bo sestavljen iz naslednjih javnih metod:

- `vstaviHuffmanovoDrevoVVerizniSeznam(HuffmanovoDrevo drevo)`,
- `sestaviZacetnoVrsto(Hashtable<Character, Integer> pogostostZnakov)`,
- `najmanjsiElementVrste()`,
- `odstraniNajmanjsega()`,
- `dolzinaVrste()`.

Poleg omenjenih metod bo vseboval še privatno metodo `vstaviHuffmanovoDrevoVVerizniSeznam(HuffmanovoDrevo drevo, VerizniSeznam sez)` in globalni objekt `VerizniSeznam seznam`. Objekt `seznam` bo tipa `VerizniSeznam`, v njem bomo hranili Huffmanova drevesa. Na začetku bomo v seznam vstavili znake in frekvence znakov, torej bo seznam sestavljen iz Huffmanovih dreves, ki bodo imela le eno vozlišče. Na koncu pa bo vseboval en element, in sicer Huffmanovo drevo.

```
import java.util.*;
public class VrstaSPrednostjo
{

  //glavni seznam dreves
  VerizniSeznam seznam = null;

  public void vstaviHuffmanovoDrevoVVerizniSeznam(
         HuffmanovoDrevo drevo)
  {
    //implementacija metode, ki v verižni seznam vstavi Huffmanovo
    //drevo
  }

  public void sestaviZacetnoVrsto(Hashtable<Character, Integer>
         pogostostZnakov)
  {
    //implementacija metode, ki sestavi vrsto s prednostjo
  }

  public HuffmanovoDrevo najmanjsiElementVrste()
  {
    //implementacija metode, ki poišče najmanjši element v vrsti
  }

  private VerizniSeznam vstaviHuffmanovoDrevoVVerizniSeznam
         (HuffmanovoDrevo drevo, VerizniSeznam sez)
  {
     //implementacija pomožne privatne metode, ki v želen verižni
    //seznam vstavi Huffmanovo drevo
  }

  public void odstraniNajmanjsega(HuffmanovoDrevo drevo)
  {
    //implementacija metode, ki izbriše najmanjši element iz vrste
  }

  public int dolzinaVrste()
  {
    //implementacija metode, ki vrne število elementov v seznamu
  }

}
```

Oglejmo si implementacijo zgoraj omenjenih metod.

Metoda `vstaviHuffmanovoDrevoVVerizniSeznam(HuffmanovoDrevo drevo)` bo tipa void. Kot vhodni parameter bo vzela objekt tipa `HuffmanovoDrevo` in ga vstavila v `VerizniSeznam seznam`.

```
public void vstaviHuffmanovoDrevoVVerizniSeznam
    (HuffmanovoDrevo drevo)
{
  //naredimo nov verižni seznam, v katerega bomo vstavili Huffmanovo
  //drevo
  VerizniSeznam seznamDreves = new VerizniSeznam();
  seznamDreves.drevo = drevo;
  seznamDreves.naslednji = seznam;
  seznam = seznamDreves; //popravimo glavni seznam
}
```

Privatna metoda `vstaviHuffmanovoDrevoVVerizniSeznam(HuffmanovoDrevo drevo, VerizniSeznam sez)` nam bo v pomoč pri implementaciji metode, ki odstrani najmanjše drevo iz seznama `seznam`. Podobna bo metodi `vstaviHuffmanovoDrevoVVerizniSeznam(HuffmanovoDrevo drevo)`. Razlika med metodama bo v tem, da bo prva metoda popravila želeni seznam dreves in ga bo vrnila, medtem ko bo druga metoda popravila globalni objekt `seznam`.

```
private VerizniSeznam vstaviHuffmanovoDrevoVVerizniSeznam
    (HuffmanovoDrevo drevo, VerizniSeznam sez)
{
  //naredimo nov verižni seznam, v katerega bomo vstavili Huffmanovo
  //drevo
  VerizniSeznam seznamDreves = new VerizniSeznam();
  seznamDreves.drevo = drevo;
  seznamDreves.naslednji = sez;
  sez = seznamDreves; //popravimo željeni seznam
  return sez;
}
```

Metoda `sestaviVrsto(Hashtable<Character, Integer> pogostostZnakov)` bo tipa void. Kot vhodni parameter bo vzela tabelo znakov in frekvenc, ki bo tipa `Hashtable`. Ključi tabele bodo `znaki`, ki bodo tipa `char`, vrednosti pa bodo `frekvence znakov`, ki bodo tipa `int`. Za vsak znak iz tabele `pogostostZnakov` bomo naredili novo vozlišče in ga vstavili v `seznam` (vrsto s prednostjo).

```
public void sestaviZacetnoVrsto(Hashtable<Character, Integer>
    pogostostZnakov)
{
  //vzamemo vse znake iz tabele pogostostZnakov in jih shranimo v
  //objekt Enumeration
  //to naredimo zato, da bomo lahko v tabeli pogostostZnakov iskali
  //znake
  Enumeration<Character> znaki = pogostostZnakov.keys();
  //dokler imamo še kaj znakov v objektu Enumeration ponavljamo
  while(znaki.hasMoreElements())
  {
    //vzamemo znak
    char znak = znaki.nextElement();
    //naredimo novo Huffmanovo vozlišče
    HuffmanovoVozlisce vozlisce = new HuffmanovoVozlisce();
    vozlisce.frekvenca = pogostostZnakov.get(znak);
    vozlisce.znak = znak;

    //Huffmanovo vozlišče vstavimo v Huffmanovo drevo
    HuffmanovoDrevo drevo = new HuffmanovoDrevo();
    //na začetku imamo le eno vozlišče v drevesu, ki ga v drevo
    //vstavimo kot koren
    drevo.koren = vozlisce;
    //na začetku še nimamo levega in desnega sina, zato jima nastavimo
    //vrednost null
    drevo.levo = null;
    drevo.desno = null;
    //vstavimo vozlišče v seznam
    vstaviHuffmanovoDrevoVVerizniSeznam(drevo);
  }
}
```

Metoda `najmanjsiElementVrste()` bo kot izhodni parameter vrnila objekt tipa `HuffmanovoDrevo`. Za sestavljeni seznam dreves `seznam`, bo poiskala najmanjše drevo v seznamu in ga vrnila.

```
public HuffmanovoDrevo najmanjsiElementVrste()
{
  //inicializiramo pomožna verižna seznama
  VerizniSeznam pom1 = seznam;
  VerizniSeznam pom2 = null;
  //nastavimo spremenljivko frekvencaMin na največjo vrednost
  int frekvencaMin = Integer.MAX_VALUE;

  //dokler imamo še kaj elementov v verižnem seznamu ponavljamo
  while(pom1 != null)
  {
    //pogledamo frekvenco v korenu Huffmanovega drevesa
    if (pom1.drevo.koren.frekvenca < frekvencaMin)
    {
      //najmanjša frekvenca postane frekvenca v korenu
      frekvencaMin = pom1.drevo.koren.frekvenca;
      //shranimo vrednosti iz pom1 v pom2
      pom2 = pom1;
    }
    //premaknemo se na naslednje drevo
    pom1 = pom1.naslednji;
  }

  //če pomožni verižni seznam ni prazen
  if(pom2 != null)
  {
    //vrnemo Huffmanovo drevo z najmanjšo frekvenco
    return pom2.drevo;
  }
  //sicer vrnemo null
  return null;
}
```

Metoda `odstraniNajmanjsega(HuffmanovoDrevo drevo)` bo tipa void. Kot vhodni parameter bo vzela `HuffmanovoDrevo`, ki ga želimo odstraniti iz objekta `seznam`.

```
public void odstraniNajmanjsega(HuffmanovoDrevo drevo)
{
  //na začetku prepišemo elemete iz seznama v pomožni seznam
  VerizniSeznam pom = seznam;
  //inicializiramo nov seznam
  VerizniSeznam novSeznam = null;

  //dokler pomožni seznam ni prazen
  while(pom != null)
  {
    //pogledamo Huffmanovem drevo iz seznama in ga primerjamo
    //z vhodnim parametrom drevo
    if(pom.drevo != drevo)
    {
      //drevesi nista enaki, zato vstavimo drevo iz pomoznega seznama
      //v nov seznam
      novSeznam = vstaviHuffmanovoDrevoVVerizniSeznam(pom.drevo,
          novSeznam);
    }
    //če sta drevesi enaki, ne naredimo ničesar (drevo ne vsatimo v
   //nov seznam)
    //v vsakem primeru, če sta drevesi enaki ali nista enaki,
    //se premaknemo naprej po pomožnem seznamu
    pom = pom.naslednji;
  }
  //popravimo globalni seznam
  seznam = novSeznam;
}
```

Oglejmo si še zadnjo metodo v razredu `VrstaSPrednostjo`, metodo `dolzinaVrste()`. Vrnila bo število elementov v objektu `seznam`.

```
public int dolzinaVrste()
{
  //na začetku je dolžina enaka 0
  int dolzina = 0;
  //globalni seznam prepišemo v pomožni seznam
  VerizniSeznam pom = seznam;
  //dokler pomožni seznam ni prazen
  while(pom != null)
  {
    //povečamo dolžino
    dolzina = dolzina +1;
    //premaknemo se naprej po pomožnem seznamu
    pom = pom.naslednji;
  }

  //vrnemo dolžino seznama
  return dolzina;
}
```

Razred `HuffmanovoKodiranje`

Razred `HuffmanovoKodiranje` bo vseboval konstruktor `HuffmanovoKodiranje(String x)` in naslednje javne metode:

- `vrniPogostostZnakovVX(String x)`,
- `zakodiraj(String  x)`,
- `odkodiraj(String y, HuffmanovoDrevo voz)`,

- `zakodirajZnake(Hashtable<Character, Integer> pogostostZnakov, HuffmanovoDrevo drevo, String geslo)`.

Vseboval bo tudi statično razredno spremenljivko `pomZnak`, ki bo tipa `char`. Predstavljala bo pomožni znak oziroma znak '¤', ki ga bomo vstavili v vsa notranja vozlišča in koren. Ta znak smo uporabili zato, da lažje in enostavnejše gradimo drevo. Poleg te spremenljivke bo razred vseboval še globalne objekte. Objekt `kodeZnakov` bo tipa `Hashtable`, v katerem bomo hranili znake in njihove binarne kode. Ključi bodo `znaki` in bodo tipa `char`, vrednosti pa `kode znakov` in bodo tipa `String`. Poleg tega bo razred kot globalno spremenljivko vseboval `HuffmanovoVozlisce`.

```
import java.util.*;

public class HuffmanovoKodiranje
{
  //Huffmanovo drevo
  public HuffmanovoDrevo drevo;
  //tabela znakov in njihovih kod
  public Hashtable<Character, String> kodeZnakov;
  //v notranjih vozliščih in korenu drevesa je znak enak pomožnemu
  //znaku
  private static char pomZnak = '¤';

  public HuffmanovoKodiranje(String x)
  {
    //konstruktor
  }

  public Hashtable<Character, Integer> vrniPogostostZnakovVX(String x)
  {
    //implementacija metode, ki vrne tabelo znakov in njihovo
    //frekvenco v nizu X
  }

  private void zakodirajZnake(Hashtable<Character, Integer>
          pogostostZnakov, HuffmanovoDrevo drevo, String koda)
  {
    //privatna metoda, ki zakodira znake
  }

  public String zakodiraj(String  x)
  {
    //implementacija metode, ki zakodira dani niz X v niz Y
  }

  public String odkodiraj(String y)
  {
    //implementacija metode, ki odkodira niz Y
  }

}
```

Najprej si oglejmo metodo, ki izračuna pogostost znakov v nizu `X` in metodo, ki zakodira znake, nato konstruktor `HuffmanovoKodiranje`, na koncu pa še metodo, ki zakodira niz `X` v niz `Y` in metodo, ki odkodira zakodirani niz `Y`.

Metoda `vrniPogostostZnakovVX(String x)` bo kot vhodni parameter vzela dani niz `X`. Za vsak različen znak iz niza `X` bo izračunala koliko krat se pojavi v nizu `X`. Vrnila bo tabelo znakov tipa `Hashtable` v kateri bo za vsak različen znak iz niza `X` zapisana njegova frekvenca. Metoda je lahko javna, saj jo lahko uporabimo "tudi tako".

```
public Hashtable<Character, Integer> vrniPogostostZnakovVX(String x)
{
  //naredimo nov objekt tipa hash table v katerega bomo shranjevali
  //znake in njihovo pogostost v nizu X
  Hashtable<Character, Integer> pogostostZnakov = new
    Hashtable<Character, Integer>();
  //dokler je še kaj znakov v nizu X
  for(int i = 0; i < x.length(); i++)
  {
    //vzamemo i-ti znak
    char znak = x.charAt(i);
    //pogledamo ali tabela pogostostZnakov že vsebuje i-ti znak
    if(pogostostZnakov.containsKey(znak))
    {
      //tabela vsebuje i-ti znak
      //zato pogostost i-tega znaka v nizu X povečamo za ena
      int pogostost =
        Integer.parseInt(String.valueOf(pogostostZnakov.get(znak)));
      pogostost = pogostost + 1;
      pogostostZnakov.put(znak, pogostost);
    }
    else
    {
      //tabela ne vsebuje i-tega znaka, zato dodamo znak v tabelo
      //pogostost znaka je enaka 1
      pogostostZnakov.put(znak, 1);
    }
  }

  //vrnemo tabelo znakov in frekvenc (pogostosti znakov)
  return pogostostZnakov;
}
```

Metoda `zakodirajZnake(Hashtable<Character, Integer> pogostostZnakov, HuffmanovoDrevo drevo, String koda)` bo dodelila kodo znakom, ki jih hranimo v tabeli `pogostostZnakov` in jih bo shranila v globalni objekt `kodeZnakov`.

```
private void zakodirajZnake(Hashtable<Character, Integer>
    pogostostZnakov, HuffmanovoDrevo drevo, String koda)
{
  //pomozna niza, ki bosta hranila kode
  String kodaLevo, kodaDesno;
  //nastavimo kodo v vozliscu
  String koda1 = koda;

  //vzamemo vse znake iz tabele pogostostZnakov in jih shranimo v
  //objekt Enumeration
  //to naredimo zato, da bomo lahko v tabeli pogostostZnakov iskali
  //znake
  Enumeration<Character> znaki = pogostostZnakov.keys();
  //če sta levo in desno poddrevo prazna, smo prišli do listov
  if(drevo.levo == null && drevo.desno == null)
  {
    //dokler imamo še kaj znakov v objektu Enumeration ponavljamo
    while(znaki.hasMoreElements())
    {
      //vzamemo znak
      char znak = znaki.nextElement();
      //primerjamo znak v korenu drevesa z znakom iz tabele
      //pogostostZnakov
      if(drevo.koren.znak == znak)
      {
        //če sta enaka, dodelimo znaku kodo
        kodeZnakov.put(znak, koda1);
      }
    }
    //smo končali s primerjanji
    return;
  }

  //še nismo prišli do listov

  //levemu sinu dodamo kodo 0
  kodaLevo = koda1 + "0";
  zakodirajZnake(pogostostZnakov, drevo.levo, kodaLevo);

  //desnemu sinu dodamo kodo 1
  kodaDesno = koda1 + "1";
  zakodirajZnake(pogostostZnakov, drevo.desno, kodaDesno);
}
```

Konstruktor `HuffmanovoKodiranje(String x)` bo kot vhodni parameter vzel dani niz `X`. Za vsak znak iz niza `X` bomo poiskali pogostost znaka v nizu `X`, nato bomo sestavili Huffmanovo drevo, na koncu bomo znakom v drevesu dodali kode.

```
public HuffmanovoKodiranje(String x)
{
  //nov objekt tipa Hashtable, ki bo hranil znake in njihove kode
  kodeZnakov = new Hashtable<Character, String>();
  //tabela znakov in njihovih frekvenc
  Hashtable<Character, Integer> pogostostZnakov =
    vrniPogostostZnakovVX(x);
  //vrsta, ki bo vsebovala vsa drevesa
  VrstaSPrednostjo vrsta = new VrstaSPrednostjo();
  //na začetku hranimo v vrsti znake iz tabele pogostostZnakov
  //drevesa v vrsti so kar listi
  vrsta.sestaviZacetnoVrsto(pogostostZnakov);

  //dokler vrsta ne vsebuje samo eno drevo ponavljamo
  while(vrsta.dolzinaVrste() > 1)
  {
    //naredimo novo HuffmanovoDrevo
    drevo = new HuffmanovoDrevo();
    //poiščemo najmanjši element vrste in ga shranimo v levo poddrevo
    drevo.levo = vrsta.najmanjsiElementVrste();
    //odstranimo najmanjši element iz vrste
    vrsta.odstraniNajmanjsega(drevo.levo);

    //poiščemo naslednji najmanjši element vrste in ga shranimo v
    //desno poddrevo
    drevo.desno = vrsta.najmanjsiElementVrste();
    //odstranimo najmanjši element iz vrste
    vrsta.odstraniNajmanjsega(drevo.desno);

    //sedaj sestavimo še koren drevesa
    drevo.koren = new HuffmanovoVozlisce();
    drevo.koren.znak = pomZnak;
    //frekvenca v vozlišču je seštevek frekvenc v sinovih vozlišča
    drevo.koren.frekvenca = drevo.levo.koren.frekvenca +
      drevo.desno.koren.frekvenca;

    //vstavimo novo Huffmanovo drevo v vrsto s prednostjo
    vrsta.vstaviHuffmanovoDrevoVVerizniSeznam(drevo);
  }

  // v drevo zapišemo dvojiške kode
  zakodirajZnake(pogostostZnakov, drevo, "");
}
```

Metoda `zakodiraj(String x)` bo kot vhodni parameter vzela dani niz `X`. S pomočjo tabele, v kateri hranimo znake in njihove kode, bo dani niz `X` pretvorila v kodirani zapis `Y` in ga vrnila.

```
public String zakodiraj(String x)
{
  //inicializirami zakodirani niz Y
  String y = "";

  //sprehodimo se po nizu x
  for(int i = 0; i < x.length(); i++)
  {
    //vzamemo i-ti znak iz niza x
    char znak = x.charAt(i);
    //pogledamo kodo znaka in jo zapišemo v niz y
    y = y + kodeZnakov.get(znak);
  }

  //vrnemo niz x v binarnem zapisu
  return y;
}
```

Ostane nam še metoda, ki zna odkodirati niz `Y`. Kot vhodni parameter bo vzela zakodirani niz `Y` in Huffmanovo drevo. Vrnila bo odkodirani niz. Seveda pričakujemo, da bo dobljeni niz enak tistemu, katerega kodiranje je Y.

```
public String odkodiraj(String y, HuffmanovoDrevo drevo)
{
  String nizX = "";
  //vzamemo zgrajeno drevo
  HuffmanovoDrevo voz = drevo;

  //dokler je še kaj znakov v nizu y ponavljamo
  while(y.length() != 0)
  {
    //če je znak v korenu drevesa enak pomožnemu znaku,
    //pomeni, da nismo še prišli do lista
    if(voz.koren.znak == pomZnak)
    {
      //pogledamo prvi znak v nizu y
      //če je znak enak 0, se premaknemo v levo poddrevo,
      //sicer pa v desno poddrevo
      if(y.charAt(0) == '0')
      {
        voz = voz.levo;
      }
      else
      {
        voz = voz.desno;
      }
      //v vsakem primeru odrežemo prvi znak iz niza y
      y = y.substring(1);
    }
    //če znak v korenu drevesa ni enak pomožnemu znaku,
    //smo prišli do listov
    else
    {
      //zapomnimo si znak
      nizX = nizX + voz.koren.znak;
      //gremo naprej odkodirat ostale znake
      voz = drevo;
    }
  }
  //v odkodiran niz zapišemo še zadnji znak
  nizX = nizX + voz.koren.znak;

  //vrnemo odkodirani niz
  return nizX;
}
```

Razred `HuffmanovoKodiranjeUporaba`
Oglejmo si uporabo zgoraj opisanih metod.

```
public class HuffmanovoKodiranjeUporaba
{
  public static void main(String [ ] args)
  {
    //dani niz X
    String x = "anja";

    //Naredimo nov objekt tipa HuffmanovoKodiranje
    HuffmanovoKodiranje huff = new HuffmanovoKodiranje(x);

    //Izpišemo znake in njihove frekvence
    System.out.println("Pogostost znakov za dani niz x: ");
    System.out.println(huff.vrniPogostostZnakovVX(x).toString());

    //izpišemo znake in njihove kode
    System.out.println("Kode znakov: ");
    System.out.println(huff.kodeZnakov.toString());

    //zakodiramo dani niz x
    String y = huff.zakodiraj(x);
    System.out.println("Zakodiran niz X: "+y);

    //odkodiramo niz y
    String odkodiranY = huff.odkodiraj(y, huff.drevo);
    System.out.println("Odkodiran niz Y: "+odkodiranY);
  }
}
```

Poglejmo si kako bi lahko izboljšali algoritem.
Vrsto s prednostjo bi lahko bolj pametno implemetirali s pomočjo podatkovne strukture minimalna kopica. V tem primeru bi popravili metode v razredu `VrstaSPrednostjo`. Popraviti bi morali metodo, ki vstavi Huffmanovo drevo v verižni seznam, metodo, ki poišče najmanjši element v verižnem seznamu in metodo, ki odstrani najmanjši element iz verižnega seznama.
Opišimo, kako bi lahko implementirali omenjene metode s pomočjo minimalne kopice in primerjavo z našo imlementacijo.

Metoda, ki vstavi element v verižni seznam.
V dani verižni seznam vstavimo Huffmanovo drevo. Metoda, ki smo jo napisali stori to tako, da Huffmanovo drevo vstavi kar na začetek verižnega seznama. S pomočjo minimalne kopice pa bi Huffmanovo drevo vstavili v verižni seznam tako, da bi imeli drevesa v seznamu urejena od največjega do najmanjšega (če rečemo, da je najmanjše drevo tisto, ki ima v korenu najmanjšo frekvenco, največje pa tisto, ki ima v korenu največjo frekvenco).

Metoda, ki poišče najmanjši element v verižnem seznamu.
Iz verižnega seznama poiščemo Huffmanovo drevo, ki ima v korenu najmanjšo frekvenco. Naša metoda stori to tako, da se sprehaja po verižnem seznamu in primerja vrednosti v korenu dreves. Če bi razred implementirali s pomočjo minimalne kopice, ne bi imeli preveč dela, saj bi vzeli kar prvo drevo iz verižnega seznama.

Metoda, ki odstrani najmanjši element iz verižnega seznama.
Iz verižnega seznama odstranimo Huffmanovo drevo, ki ima v korenu najmanjšo frekvenco. V naši metodi se ponovno sprehodimo čez celoten verižni seznam in iščemo drevo, ki je enako drevesu, ki ga želimo odstraniti iz verižnega seznama. S pomočjo minimalne kopice pa bi enostavno odstranili prvi element v seznamu.

Metoda, ki smo jo napisali za vstavljanje elementa v verižni seznam je hitrejša od implementacije iste metode s pomočjo minimalne kopice. Sta pa zato metodi za brisanje in iskanje najmanjšega elementa precej

zahtevni. Če bi ju implementirali s pomočjo minimalne kopice bi bili zelo hitri, saj vzameta kar prvi element iz verižnega seznama.

2.1.4 Časovna zahtevnost

Oglejmo si časovno zahtevnost algoritma Huffmanovo kodiranje.
Omenili smo, da smo vrsto s prednostjo neučinkovito implementirali. Zato si najprej poglejmo časovno zahtevnost algoritma s slabo implementirano vrsto s prednostjo, nato pa poglejmo kako bi lahko vrsto s prednostjo boljše implementirali in s tem prihranili pri časovni zahtevnosti algoritma.

Algoritem je sestavljen iz treh delov. V prvem delu vzamemo znake iz abecede `A`, ki vsebuje vse različne znake iz danega niza `X` in zgradimo Huffmanovo drevo. Torej za vsak znak iz abecede `A` izračunamo pogostost pojavitve v nizu `X`. Pri tem opravimo natanko toliko primerjanj, kolikor je dolžina niza `X`, torej `n`. Zato potrebujemo natanko `O(n)` časa. Nato inicializiramo vrsto s prednostjo, v katero na začetku vstavimo vse znake iz abecede `A`. Za to potrebujemo `O(d)` časa, kjer je `d` dolžina abecede `A`. Dokler vrsta s prednostjo ne vsebuje samo enega elementa, vzamemo dve drevesi z najmanjšima frekvencama in ju združimo v skupno drevo, ki ga vstavimo v vrsto s prednostjo. Drevesi pa odstranimo iz vrste. Vsaka taka operacija je odvisna od dolžine verižnega seznama Huffmanovih dreves. Označimo dolžino verižnega seznama z `dvs`. Na začetku je dolžina verižnega seznama enaka d. Torej bo vsaka taka operacija zahtevala `O(dvs)` časa. To pa izvedemo `d - 1` krat (koliko krat se izvede while zanka). Torej `O(d * dvs)`.
Na koncu zgrajenemu drevesu dodamo dvojiške kode. Za to porabimo toliko časa kolikor imamo povezav v drevesu. Če je vozlišč v drevesu enako `v`, je povezav v drevesu enako `v - 1`. Torej porabimo `O(v - 1)` časa.

Predvidena časovna zahtevnost prvega dela algoritma (torej gradnja drevesa) Huffmanovo kodiranje je:

$$T_{H1}(n, d, dvs, v) = O(n) + O(d * dvs) + O(v - 1) = O(n + d * dvs + v)$$

Pri gradnji Huffmanovega drevesa je časovna zahtevnost dejansko odvisna od dolžine abecede A. Zato pri gradnji drevesa nimamo najboljšega oziroma najslabšega primera.

Drugi del algoritma nastopi, ko želimo dani niz `X` zakodirati. Torej celoten niz `X` zapišemo z ničlami in enkami. Kode znakov imamo shranjene v tabeli `kodeZnakov`. Torej moramo narediti še naslednje, sprehodimo se po nizu `X`, vzamemo trenutni znak iz niza `X`, pogledamo njegovo kodo in si jo zapomnemo v nizu `y`. Za to porabimo natanko `O(n)` časa.

Predvidena časovna zahtevnost drugega dela algoritma Huffmanovo kodiranje je:

$$T_{H2}(n) = O(n)$$

V najboljšem in najslabšem primeru je časovna zahtevnost drugega dela algoritma še vedno odvisna od dolžine danega niza `X`. Torej bo v najboljšem in najslabšem primeru časovna zahtevnost še vedno enaka `O(n)`.

Zadnji del algoritma vsebuje odkodiranje niza `y`. Ničle in enke v nizu `y` zamenjamo z znaki. Dokler imamo še kaj znakov v nizu `y` vzamemo prvi znak. Če je znak enak `'0'`, sledimo povezavi iz vozlišča v levega sina, sicer sledimo povezavi iz vozlišča v desnega sina. Režemo znake v nizu `y`. Ko pridemo do lista v drevesu, si zapomnimo znak v listu. To opravimo v `O(m)` času. Pri tem je `m` dolžina niza `y`

Predvidena časovna zahtevnost tretjega dela algoritma Huffmanovo kodiranje je:

$$T_{H3}(m) = O(m)$$

Tudi pri odkodiranju dejansko nimamo najboljšega in najslabšega primera. Časovna zahtevnost je odvisna od dolžine niza Y, ne pa od podatkov, ki jih hranimo v nizu Y.

Poglejmo si časovno zahtevnost algoritma, če bi vrsto s prednostjo implementirali s pomočjo minimalne kopice. Pri časovni zahtevnosti bi pridobili pri gradnji Huffmanovega drevesa. Vsaka taka operacija, ki poišče najmanjši element v vrsti, ga odstrani iz vrste in vstavi novo nastalo Huffmanovo drevo porabi `O(logd)` časa, kjer je `d` dolžina abecede `A`. Torej bi bila pričakovana skupna časovna zahtevnost prvega dela algoritma enaka

$$T_{H1}(n, d, v) = O(n) + O(logd) + O(v - 1) = O(n + logd + v)$$

3 Preverjanje podobnosti besedila (Text Similarity Testing)

V genetiki in programerskem inženiringu se pogosto srečujemo s problemom preverjanja podobnosti dveh nizov. V genetiki to preverjanje uporabljamo, kadar iščemo sorodnost med dvema individualnima subjektoma. Niza sta v tem primeru DNA zapisa.
V programerskem inženiringu pogosto primerjamo kodo, na kateri trenutno delamo in njeno predhodno verzijo. Niza sta v tem primeru vsebina datoteke, ki vsebuje kodo, na kateri trenutno delamo in vsebina datoteke, ki vsebuje njeno predhodno verzijo.

3.1.1 Problem najdaljšega skupnega podzaporedja (LCS)

Denimo, da imamo podan niz $X = x_0\ x_1\ x_2\ \ldots\ x_{n-1}$. Podzaporedje danega zaporedja X je vsak niz oblike x_{i_1}, x_{i_2}, …, x_{i_k}, ki ustreza pogoju $i_j < i_{j+1}$.

Primer:
Vzemimo zaporedje "ACBDEGCEDBG". Podzaporedje danega zaporedja je "BCDG", ki ustreza indeksnemu zaporedju 3, 7, 9, 11.
Kot vidimo, beseda podzaporedje ne pomeni isto kot beseda podniz (substring).

Problem, ki ga definira LCS (longest common subsequence), je naslednji:
Dana imamo dva niza znakov, $X = x_0\ x_1\ x_2\ \ldots\ x_{n-1}$ in $Y = y_0\ y_1\ y_2\ \ldots\ y_{m-1}$. Znaki iz X in Y ustrezajo znakom iz določene abecede A. Najti moramo najdaljši niz S, ki bo hkrati podzaporedje v X in v Y.

Primeri:
Dan imamo niz X = "manjšina" in niz Y = "manjše". Iščemo najdaljši niz S, ki bo hkrati podzaporedje v X in v Y. V našem primeru je S = "manjš". Dolžina niza S je enaka 5.
Za dana niza X = "televizija" in Y = "telefonija" je najdaljše podzaporedje S = "teleija". Dolžina niza S je enaka 7. Za niza X = "diploma" in Y = "zaposlen" pa je najdaljše skupno podzaporedje S = "po".

Naivni pristop reševanja LCS problema je, da pregledamo vsa podzaporedja v X in vzamemo najdaljše podzaporedje, ki je tudi v Y. Da ugotovimo, ali je nek niz podzaporedje v drugem nizu, lahko uporabimo algoritem, ki je opisan v razdelku 3.1.5.1 Naivni pristop. A v nizu X imamo kar 2^n različnih podzaporedij. Algoritem, ki ugotovi, ali je neko podzaporedje tudi podzaporedje v Y, je časovne zahtevnosti O(m). Pri tem je m dolžina niza Y. Časovna zahtevnost naivnega pristopa je $O(2^n * m)$. Algoritem je torej zelo počasen. V tem poglavju si bomo ogledali, kako problem LCS lahko rešimo hitreje s pomočjo algoritma, ki uporablja tehniko, imenovano dinamično programiranje.

3.1.2 Dinamično programiranje

Dinamično programiranje je metoda, ki sistematično pregleduje možne poti pri reševanju danega problema in pripelje do optimalne rešitve. Uporabimo jo takrat, ko pri naivnem reševanju problema (z algoritmom Groba sila - pregledovanjem vseh možnosti) ugotovimo, da bo časovna zahtevnost eksponentnega reda.

Dinamično programiranje običajno reši problem hitreje, saj nam pogosto uspe razviti algoritem, kjer je časovna zahtevnost polinomskega reda.

Osnovna ideja reševanja problema s pomočjo dinamičnega programiranja je, da problem razdelimo na več podproblemov, ki so enostavnejši za reševanje in so med seboj povezani. Običajno so ti podproblemi iste vrste kot glavni problem. Iz rešitev podproblemov sestavljamo rešitev glavnega problema. Rešitev glavnega problema je optimalna, zato morajo biti vse delne rešitve (rešitve podproblemov) tudi optimalne. Temu rečemo pravilo optimalnosti.

Koraki reševanja problema s pomočjo dinamičnega programiranja:

- Če je glavni problem enostavno rešljiv, ga neposredno rešimo. Sicer ga razdelimo na več enostavnejših podproblemov, ki so običajno istega tipa kot glavni problem in poiščemo njihove optimalne rešitve.
- Podprobleme rešujemo ločeno. Zapomnimo si rešitve podproblemov.
- Optimalno rešitev problema sestavimo iz optimalnih rešitev podproblemov.

3.1.3 Reševanje LCS problema s pomočjo dinamičnega programiranja

Spomnimo se, da rešujemo naslednji problem:
Dana imamo dva niza znakov, $X = x_0\ x_1\ x_2\ \ldots\ x_{n-1}$ in $Y = y_0\ y_1\ y_2\ \ldots\ y_{m-1}$. Znaki ustrezajo znakom iz abecede A. Med vsemi nizi S, ki so hkrati podzaporedje v X in v Y, moramo poiskati najdaljšega.

Iskanje najdaljšega skupnega podzaporedja bomo razdelili na dva dela. V prvem delu bom pokazali, kako zgradimo matriko L, ki bo vsebovala dolžine skupnih podzaporedij delov predpon nizov X in Y (torej začetkov nizov X in Y). V drugem delu pa bomo s pomočjo matrike L ugotovili, katero je tisto najdaljše skupno podzaporedje, ki je hkrati podzaporedje niza X in Y.

3.1.3.1 Gradnja matrike L

Z L[i, j] definiramo dolžino najdaljšega niza, ki je podzaporedje tako v nizu $X[1 \ldots i] = x_1\ x_2\ x_3\ \ldots\ x_i$ kot v nizu $Y[1 \ldots j] = y_1\ y_2\ y_3\ \ldots\ y_j$.

Poglejmo, kako bi L[i, j] lahko izračunali. Predstavljajmo si niz X[1 … i] sestavljen iz dveh delov. Prvi del sestavlja začetnih i - 1 znakov ($x_1\ x_2\ x_3\ \ldots\ x_{i-1}$), drugi del pa znak x_i. Podobno razdelimo niz Y[1 … j], in sicer na $y_1\ y_2\ y_3\ \ldots\ y_{j-1}$ in y_j. Primerjajmo znaka x_i in y_j. Če sta enaka, je očitno L[i, j] = L[i - 1, j - 1] + 1. Dokažimo to! Naj bo s* najdaljši skupni podniz v X[1 ... i - 1] in Y[1 ... j - 1]. Njegova dolžina je po definiciji L[i - 1, j - 1]. Če mu dodamo x_i, je to očitno podniz v X[1 ... i]. Ker pa je podniz s^*x_i isti niz kot s^*y_j, je to hkrati tudi podniz v Y[1 … j]. Zagotovo je to najdaljši skupni podniz. Če bi namreč bil kakšen drug podniz daljši, bi moral biti daljši že od s*, kar pa po predpostavki ni res!
Kaj pa če sta znaka x_i in y_j različna? Potem je L[i, j] = max{L[i - 1, j], L[i, j - 1]}. V tem primeru ne bomo imeli najdaljšega skupnega podzaporedja, ki bi vsebovalo x_i in y_j, ampak lahko vsebuje x_i ali y_j, lahko pa nobenega od njiju.

Zapišimo ugotovitev:

$$L[i,j] = \begin{cases} L[i-1, j-1] + 1 & x_i = y_j \\ \max\{L[i-1, j], L[i, j-1]\} & x_i \neq y_j \end{cases}$$

Za začetne vrednosti lahko nastavimo L[i, 0] = L[0, j] = 0, saj ustreznih podzaporedij ni. Zato je smiselno, da dolžino nastavimo na 0.

Iščemo seveda L[n, m]. Poglejmo, kako jo izračunamo. V ta namen bomo potrebovali vse vrednosti L[i, j], 0 <= i <= n, 0 <= j <= m. Zato si pripravimo matriko dimenzije (n + 1) x (m + 1). Pokažimo sedaj, kako izračunamo to matriko.

Naj bo X = "predvidevanje" in Y = "revizija". Dolžina niza X je enaka n = 13, Y pa m = 8. Matrika bo torej dimenzije 14 x 9. Stolpce matrike bo sestavljal indeks j, ki teče po nizu Y, vrstice pa indeks i, ki teče po nizu X. Da bomo dobili občutek za gradnjo matrike L, najprej napišimo vse vrednosti matrike, nato pa za del matrike L pokažimo, kako smo vrednosti izračunali.

			r	e	v	i	z	i	j	a
	L	0	1	2	3	4	5	6	7	8
	0	0	0	0	0	0	0	0	0	0
p	1	0	0	0	0	0	0	0	0	0
r	2	0	1	1	1	1	1	1	1	1
e	3	0	1	2	2	2	2	2	2	2
d	4	0	1	2	2	2	2	2	2	2
v	5	0	1	2	3	3	3	3	3	3
i	6	0	1	2	3	4	4	4	4	4
d	7	0	1	2	3	4	4	4	4	4
e	8	0	1	2	3	4	4	4	4	4
v	9	0	1	2	3	4	4	4	4	4
a	10	0	1	2	3	4	4	4	4	5
n	11	0	1	2	3	4	4	4	4	5
j	12	0	1	2	3	4	4	4	5	5
e	13	0	1	2	3	4	4	4	5	5

Poglejmo si kako smo izračunali vrednosti matrike za i = 9 in j = 6, 7, 8. Torej računamo zadnje tri stolpce v deveti vrstici matrike. Na sliki označimo, katere vrednosti bomo naračunali.

		…	z	i	j	a
	L	…	5	6	7	8
…	…	…	…	…	…	…
e	8	…	4	4	4	4
v	9	…	4	?	?	?
a	10	…	4			
n	11	…	4			
j	12	…	4			
e	13	…	4			

Najprej si poglejmo, kako izračunamo vrednost L[9, 6]. Primerjamo deveti znak iz niza X, znak 'v' in šesti znak iz niza y, znak 'i'. Ker znaka nista enaka, moramo preveriti, kakšna je dolžina najdaljšega skupnega podzaporedja prvih devet znakov iz niza X, to je niz "predvidev" in prvih pet znakov iz niza Y, to je niz "reviz". Pogledamo torej L[9, 5]. S tem smo preverili, če se deveti znak iz niza X "splača" vključiti. Po drugi strani pa je možno še, da se bolj splača vzeti šesti znak iz Y. Torej preverimo kakšna je dolžina najdaljšega skupnega podzaporedja prvih osem znakov iz X ("predvide") in prvih šest znakov iz Y ("revizi"). Pogledamo torej L[8, 6]. Vzamemo večjo vrednost od vrednosti L[9, 5] = 4 in L[8, 6] = 4. Ker sta vrednosti enaki, je vseeno katero vzamemo. Iz tega sledi, da bo vrednost L[9, 6] = 4.

		…	z	i	…
	L	…	5	6	…
…	…	…	…	…	…
e	8	…	4	4	4
v	9	…	4	4	…
…	…	…	…	…	…

V tabeli sta z modro barvo označeni vrednosti, med katerima zbiramo, z rdečo pa dolžina najdaljšega skupnega podzaporedja v nizih "predvidev" in "revizi".

Izračunajmo naslednjo vrednost v matriki L, vrednost L[9, 7]. Primerjamo znaka 'v' in 'j'. Znaka nista enaka. Sedaj pogledamo vrednosti L[9, 6] in L[8, 7]. To pomeni, da pogledamo, ali je daljši najdaljši skupni niz med prvimi devetimi znaki iz niza X in šestimi v nizu Y, ali pa je daljše najdaljše podzaporedje med prvimi osmimi znaki iz Y in sedmimi znaki iz niza Y. Ker je dolžina najdaljšega skupnega podzaporedja niza "predvidev" in "revizi" enaka dolžini najdaljšega skupnega zaporedja niza "predvide" in "revizij", je vseeno, katero vrednost vzamemo. V vsakem primeru bo L[9, 7] = 4.

		…	z	i	j
	L	…	5	6	7
…	…	…	…	…	…
e	8	…	4	4	4
v	9	…	4	4	4
…	…	…	…	…	…

Vrednost L[9, 8] bo tudi enaka 4. Znaka 'v' in 'a' nista enaka. Torej vzamemo večjo vrednost med L[9, 7] in L[8, 8].

		…	z	i	j	a
	L	…	5	6	7	8
…	…	…	…	…	…	…
e	8	…	4	4	4	4
v	9	…	4	4	4	4
…	…	…	…	…	…	…

Sedaj pa si poglejmo kako smo izračunali vrednosti L[12, 7], L[12, 8]. Torej računamo zadnja dva stolpca v dvanajsti vrstici.

		…	i	j	a
	L	…	6	7	8
…	…	…	…	…	…
n	11	…	4	4	5
j	12	…	4	?	?
e	13	…	4		

Najprej si poglejmo, kako bi izračunali L[12, 7]. Zanima nas dolžina najdaljšega skupnega podzaporedja prvih dvanajst znakov niza X ("predvidevanj") in prvih sedem znakov iz niza Y ("revizij"). Opazimo, da je dvanajsti znak iz niza X, znak 'j' enak sedmemu znaku iz niza Y, znaku 'j'. To pomeni, da bo dolžina najdaljšega skupnega podzaporedja niza "predvidevanj" in "revizij" za eno večje od najdaljšega skupnega podzaporedja prvih enajst znakov iz niza X ("predvidevan") in prvih šest znakov iz niza Y ("revizi"). Torej bo vrednost L[12, 7] = L[11, 6] + 1.

		…	i	j	a
	L	…	6	7	8
…	…	…	…	…	…
n	11	…	4	4	5
j	12	…	4	5	
e	13	…	4		

Za izračun vrednosti L[12, 8] uporabimo enak postopek, kot smo ga uporabili pri izračunu vrednosti L[9, 8]. Ker se znak 'e' iz niza X ne ujema z znakom 'a' iz niza Y, pogledamo vrednosti pri L[12, 7] in L[11, 8]. Dolžina najdaljšega skupnega podzaporedja niza "predvidevanj" in niza "revizij" (to je L[12, 7]) je enaka 5. Tudi dolžina najdaljšega skupnega podzaporedja niza "predvidevan" in niza "revizija" (to je L[11, 8]) je enaka 5. Iz tega sledi, da je L[12, 8] = 5. To pomeni, da je dolžina najdaljšega skupnega podzaporedja niza "predvidevanj" in niza "revizija" enaka 5.

		…	i	j	a
	L	…	6	7	8
…	…	…	…	…	…
n	11	…	4	4	5
j	12	…	4	5	5
e	13	…	4		

3.1.3.2 Iskanje najdaljšega skupnega podzaporedja

Poglejmo si, kako s pomočjo izračunane matrike L najdemo najdaljše skupno podzaporedje.

Zapišimo postopek iskanja najdaljšega skupnega podzaporedja.
Primerjamo znake iz niza X z znaki iz niza Y. Začnemo na koncu niza X in Y in se premikamo proti začetku. Tako ravnamo, ker vemo, da ima najdaljše skupno podzaporedje L[n, m] znakov.
Začnemo torej s celotnima nizoma X in Y. Preverimo zadnja znaka:

- Če je znak na mestu X[n] enak znaku na mestu Y[m], se premaknemo za eno v levo v obeh nizih in si zapomnimo znak. Premik v levo pomeni, da skrajšamo niz tako, da režemo zadnji znak. Popravimo dolžino niza.
- Znaka nista enaka. Zato pogledamo v matriko L, ki nam bo povedala, v katerem nizu se premaknemo v levo. Vemo, da smo L[n, m] izračunali tako, da smo vzeli max{L[n, m - 1] , L[n - 1, m]}, zato v matriki L pogledamo vrednost na mestu L[n, m - 1] in L[n - 1, m]. Če velja L[n, m - 1] < L[n - 1, m], se premaknemo za ena v levo po nizu X, sicer se premaknemo za ena v levo po nizu Y. Če sta vrednosti enaki, je vseeno po katerem nizu se bomo premaknili, mi bomo takrat šli za ena v levo po nizu Y.

Sedaj nadaljujemo na »skrajšanih« nizi X in Y. Iz zgornjega opisa je razvidno, da se bo vsaj en od nizov X in Y skrajšal za en znak.

Prikažimo zgoraj opisan postopek na primeru. Vzemimo matriko L, ki smo jo zgradili v podpoglavju 3.1.3.1 Gradnja matrike L.

			r	e	v	i	z	i	j	a
	L	0	1	2	3	4	5	6	7	8
	0	0	0	0	0	0	0	0	0	0
p	1	0	0	0	0	0	0	0	0	0
r	2	0	1	1	1	1	1	1	1	1
e	3	0	1	2	2	2	2	2	2	2
d	4	0	1	2	2	2	2	2	2	2
v	5	0	1	2	3	3	3	3	3	3
i	6	0	1	2	3	4	4	4	4	4
d	7	0	1	2	3	4	4	4	4	4
e	8	0	1	2	3	4	4	4	4	4
v	9	0	1	2	3	4	4	4	4	4
a	10	0	1	2	3	4	4	4	4	5
n	11	0	1	2	3	4	4	4	4	5
j	12	0	1	2	3	4	4	4	5	5
e	13	0	1	2	3	4	4	4	5	5

Iskanje najdaljšega skupnega zaporedja začnemo tako, da pogledamo zadnja dva znaka iz nizov X in Y. Torej pogledamo znaka 'e' in 'a'. Ker znaka nista enaka, pogledamo, ali smo našli skupno podzaporedje na mestu L[12, 8] ali na mestu L[13, 7].

		…	j	a
	L	…	7	8
…	…	…	…	…
j	12	…	5	5
e	13	…	5	5

Ker je L[12, 8] = L[13, 7], je dolžina najdaljšega skupnega podzaporedja nizov "predvidevanj" in "revizija" enaka dolžini najdaljšega skupnega podzaporedja nizov "predvidevanje" in "revizij". Skrajšali bi lahko bodisi niz X, bodisi niz Y. Kot smo povedali prej, se odločimo, da takrat vedno režemo znak v nizu Y. Novi niz Y bo enak "revizij".

			r	e	v	i	z	i	j	a
	L	0	1	2	3	4	5	6	7	8
	0	0	0	0	0	0	0	0	0	0
p	1	0	0	0	0	0	0	0	0	0
r	2	0	1	1	1	1	1	1	1	1
e	3	0	1	2	2	2	2	2	2	2
d	4	0	1	2	2	2	2	2	2	2
v	5	0	1	2	3	3	3	3	3	3
i	6	0	1	2	3	4	4	4	4	4
d	7	0	1	2	3	4	4	4	4	4
e	8	0	1	2	3	4	4	4	4	4
v	9	0	1	2	3	4	4	4	4	4
a	10	0	1	2	3	4	4	4	4	5
n	11	0	1	2	3	4	4	4	4	5
j	12	0	1	2	3	4	4	4	5	5
e	13	0	1	2	3	4	4	4	5	5

Ponovno vzamemo zadnja znaka iz niza X in Y. To sta znaka 'j' in 'j'. Znaka sta enaka. To nam pove, da bo znak 'j' zagotovo del najdaljšega skupnega podzaporedja. Zato si znak 'j' zapomnimo. Odrežemo zadnja znaka iz niza X in Y.

			r	e	v	i	z	i	j	a
	L	0	1	2	3	4	5	6	7	8
	0	0	0	0	0	0	0	0	0	0
p	1	0	0	0	0	0	0	0	0	0
r	2	0	1	1	1	1	1	1	1	1
e	3	0	1	2	2	2	2	2	2	2
d	4	0	1	2	2	2	2	2	2	2
v	5	0	1	2	3	3	3	3	3	3
i	6	0	1	2	3	4	4	4	4	4
d	7	0	1	2	3	4	4	4	4	4
e	8	0	1	2	3	4	4	4	4	4
v	9	0	1	2	3	4	4	4	4	4
a	10	0	1	2	3	4	4	4	4	5
n	11	0	1	2	3	4	4	4	4	5
j	12	0	1	2	3	4	4	4	5	5
e	13	0	1	2	3	4	4	4	5	5

Zadnji znak iz niza X je 'j', iz niza Y pa 'i'. Znaka nista enaka. Pogledamo, ali smo našli skupno podzaporedje na mestu L[12, 5] ali na mestu L[11, 6]. Ker sta vrednosti enaki, režemo zadnji znak v nizu Y.

			r	e	v	i	z	i	j	a
	L	0	1	2	3	4	5	6	7	8
	0	0	0	0	0	0	0	0	0	0
p	1	0	0	0	0	0	0	0	0	0
r	2	0	1	1	1	1	1	1	1	1
e	3	0	1	2	2	2	2	2	2	2
d	4	0	1	2	2	2	2	2	2	2
v	5	0	1	2	3	3	3	3	3	3
i	6	0	1	2	3	4	4	4	4	4
d	7	0	1	2	3	4	4	4	4	4
e	8	0	1	2	3	4	4	4	4	4
v	9	0	1	2	3	4	4	4	4	4
a	10	0	1	2	3	4	4	4	4	5
n	11	0	1	2	3	4	4	4	4	5
j	12	0	1	2	3	4	4	4	5	5
e	13	0	1	2	3	4	4	4	5	5

Po nekaj korakih pridemo do stanja:

			r	e	v	i	z	i	j	a
	L	0	1	2	3	4	5	6	7	8
	0	0	0	0	0	0	0	0	0	0
p	1	0	0	0	0	0	0	0	0	0
r	2	0	1	1	1	1	1	1	1	1
e	3	0	1	2	2	2	2	2	2	2
d	4	0	1	2	2	2	2	2	2	2
v	5	0	1	2	3	3	3	3	3	3
i	6	0	1	2	3	4	4	4	4	4
d	7	0	1	2	3	4	4	4	4	4
e	8	0	1	2	3	4	4	4	4	4
v	9	0	1	2	3	4	4	4	4	4
a	10	0	1	2	3	4	4	4	4	5
n	11	0	1	2	3	4	4	4	4	5
j	12	0	1	2	3	4	4	4	5	5
e	13	0	1	2	3	4	4	4	5	5

Do sedaj smo v skupni podniz »nabrali« le »j«. Iz L[12,4], ki je 4, razberemo, da morata imeti niza »revi« in »predvidevanj« najdaljše skupno podzaporedje dolžine 4.
Primerjamo znak 'j' iz niza X z znakom 'i' iz niza Y. Znaka nista enaka. Ker je L[12, 3] < L[11, 4], to pomeni, da je najdaljše skupno podzaporedje prvih dvanajst znakov niza X ("predvidevanj") in prvih treh znakov niza Y ("rev") krajše, kot je dolgo najdaljše skupno podzaporedje prvih enajst znakov iz niza X ("predvidevan") in prvih štirih znakov iz niza Y (revi). Zato režemo zadnji znak iz niza X.

			r	e	v	i	z	i	j	a
	L	0	1	2	3	4	5	6	7	8
	0	0	0	0	0	0	0	0	0	0
p	1	0	0	0	0	0	0	0	0	0
r	2	0	1	1	1	1	1	1	1	1
e	3	0	1	2	2	2	2	2	2	2
d	4	0	1	2	2	2	2	2	2	2
v	5	0	1	2	3	3	3	3	3	3
i	6	0	1	2	3	4	4	4	4	4
d	7	0	1	2	3	4	4	4	4	4
e	8	0	1	2	3	4	4	4	4	4
v	9	0	1	2	3	4	4	4	4	4
a	10	0	1	2	3	4	4	4	4	5
n	11	0	1	2	3	4	4	4	4	5
j	12	0	1	2	3	4	4	4	5	5
e	13	0	1	2	3	4	4	4	5	5

Postopek nadaljujemo, dokler bodisi niz X bodisi niz Y ne vsebuje nobenega znaka več. V našem primeru tako ugotovimo, da je najdaljše skupno podzaporedje nizov »revizija« in »predvidevanje« enako "revij".

3.1.4 Primeri

Primer 1:
Naj bo X = "kaj" in Y = "ja". Najprej poiščimo dolžino najdaljšega podzaporedja S, ki je hkrati podzaporedje nizu X in v nizu Y. Nato bomo to podzaporedje še izpisali.

Gradnja matrike L
Dolžina niza X je enaka n = 3, Y pa m = 2. Matrika bo torej dimenzije 4 x 3. Stolpce matrike bo sestavljal indeks j, ki teče po nizu Y, vrstice pa indeks i, ki teče po nizu X. Nastavimo vrednosti L[i, 0] = L[0, j] = 0. Pripravimo si matriko:

			j	a
	L	0	1	2
	0	0	0	0
k	1	0		
a	2	0		
j	3	0		

Sedaj začnemo s primerjanji. Vzamemo prvi znak iz X, znak 'k' in ga primerjamo z znaki iz Y, torej najprej z 'j' in nato z 'a'. Opazimo, da 'k' ≠ 'j'. Ker sta znaka različna in so predhodno "izračunane" dolžine enake 0, bo tudi L[1, 1] = 0. Enako velja za naslednji korak, kjer primerjamo znaka 'k' in 'a'. Vpišimo vrednosti v matriko:

			j	a
	L	0	1	2
	0	0	0	0
k	1	0	0	0
a	2	0		
j	3	0		

Izračunali smo prvo vrstico. Sedaj računamo vrstico, kjer primerjamo znak 'a' iz X z 'j' in 'a' iz Y. Najprej primerjamo znak 'a' z znakom 'j'. Ker sta različna in predhodno nismo našli nobenega ujemanja, dolžine so enake 0, bo tudi L[2, 1] = 0. Na naslednjem koraku pa sta znaka enaka ('a' = 'a'). Tu smo našli prvo ujemanje, zato je dolžina L[2, 2] enaka 1.

			j	a
	L	0	1	2
	0	0	0	0
k	1	0	0	0
a	2	0	0	1
j	3	0		

Ostane nam še računanje zadnje vstice, kjer primerjamo znak 'j' iz X z 'j' in 'a' iz Y. Na prvem koraku najdemo ujemanje ('j' = 'j'). Ker je L[i - 1, j - 1] = L[2, 0] = 0, bo L[3, 1] = 1, saj je to najdaljše skupno podzaporedje, ki smo ga našli do sedaj. Na koncu primerjamo še znaka 'j' in 'a'. Znaka sta različna, torej pogledamo, ali smo predhodno našli skupno podzaporedje v nizu X in Y. Ker je L[i - 1, j] = L[2, 2] = 1 in L[i, j - 1] = L[3, 1] = 1, pomeni, da smo našli skupna podzaporedja, ki sta dolžine 1. Torej bo L[3, 2] = 1.

			j	a
	L	0	1	2
	0	0	0	0
k	1	0	0	0
a	2	0	0	1
j	3	0	1	1

Ugotovili smo, da je dolžina najdaljšega skupnega podzaporedja, torej L[3, 2], enaka 1. Niza »ja« in »kaj« imata torej skupno podzaporedje dolžine 1, daljšega skupnega podzaporedja pa ne. Zanima nas, katero je to podzaporedje. Že na prvi pogled opazimo, da sta podzaporedji dve in sicer niza "a" in "j". A poglejmo, če do tega lahko pridemo tudi s pomočjo matrike L.

Sestavljanje najdaljšega skupnega podzaporedja

Vpišimo niza X = "kaj" in Y = "ja" v tabelo. Dolžina niza X je n = 3, Y pa m = 2.

X	k	a	j
Y	j	a	
j	0	1	
i	0	1	2

Korak 0:

Vzamemo zadnja znaka iz niza X in Y in ju primerjamo. Znaka 'j' in 'a' nista enaka, zato pogledamo v matriko L. Zanima nas ali smo našli skupno podzaporedje na mestu L[n, m - 1] = L[3, 1] ali na mestu L[n - 1, m] = L[2, 2]. Ker je L[3, 1] = L[2, 2] = 1, pomeni, da se bomo premaknili za ena v levo po nizu Y. Torej bo novi Y = "j".

X	k	a	j
Y	j		
j	0		
i	0	1	2

Korak 1:

Primerjamo znak 'j' iz niza X z znakom 'j' iz Y. Znaka sta enaka. Ker smo prišli do konca niza Y, je najdaljše skupno podzaporedje S = "j".

Primer 2:
Na začetku poglavja smo omenili, da se algoritem za iskanje najdaljšega skupnega podzaporedja uporablja tudi v genskem inženiringu. S pomočjo algoritma ugotavljajo, ali sta dve osebi v sorodu (ali imata podoben DNK zapis). Informacije v deoksiribonukleinski kislini (DNK) so sestavljene iz kombinacije štirih gradnikov: adenin, citozin, gvanin in timin. Lahko bi rekli, da so sestavljene iz štirih črk, in sicer A, C, G, T. V večini primerov se za prikaz delovanja algoritma uporabljajo te črke. Oglejmo si en tak primer.

Dan imamo niz X = "GTTACA" in niz Y = "TTGACAGA". Poiščimo dolžino najdaljšega podzaporedja S, ki je hkrati podzaporedje v nizu X in v nizu Y. Ko bomo dolžino določili, pa nas zanima še, katero je to podzaporedje S.

Gradnja matrike L
Na začetku indeksiramo znake v nizu X in Y in indekse vpišemo v matriko. Za lažje računanje dodamo matriki še indeks 0. Matrika bo dimenzije (n + 1) x (m + 1). Kar napišimo njene vrednosti.

			T	T	G	A	C	A	G	A
	L	0	1	2	3	4	5	6	7	8
	0	0	0	0	0	0	0	0	0	0
G	1	0	0	0	1	1	1	1	1	1
T	2	0	1	1	1	1	1	1	1	1
T	3	0	1	2	2	2	2	2	2	2
A	4	0	1	2	2	3	3	3	3	3
C	5	0	1	2	2	3	4	4	4	4
A	6	0	1	2	2	3	4	5	5	5

Iz matrike lahko preberemo, da je dožina najdaljšega podzaporedja enaka L[6, 8] = 5. Prav tako pa je seveda v matriki na voljo še druga informacija. Tako lahko razberemo, da imata niza "TTGAC" in "GTTA" (torej prvih 5 znakov niza X in prvi 4 znaki niza y) najdaljše podzaporedje dolžine 3. Prav tako vidimo, da ima prvih 5, 6, 7 in 8 znakov niza X s podnizom "GTTAC" skupno podzaporedje dolžine 4 ...

Opišimo sedaj, kako smo sestavili matriko.

Prvi stolpec in prvo vrstico (tam, kjer smo uporabili indeks 0) napolnimo z ničlami. To vrstico smo dodali zaradi lažjega računanja. Ko je i = 0 ali j = 0, pomeni, da je bodisi niz X bodisi niz Y prazen. Ker je eden od nizov prazen, skupno podzaporedje ne obstaja in zato je dolžina enaka 0.

Korak 1:
Izračunajmo 1. vrstico, torej L[1, j], za j = 1, 2, ..., 8.
V prvi vrstici nas zanima največja dolžina skupnih podnizov niza X[1] in Y[1 … j]. Ker je podniz X[1] enak "G", nas torej zanima, če v katerem od podnizov Y[1 … j] nastopa znak 'G'.

Dokler v Y ne naletimo na 'G', bodo vrednosti L[1, j] enake 0, saj v Y ni ustreznega podniza. Od mesta, kjer v Y prvič nastopa 'G', pa do konca, pa bo L[1, j] seveda 1.
Poglejmo, kako bi premislek, ki smo ga pokazali na konkretnem primeru, uporabili za računanje prve vrstice. Če dobro premislimo, je formula za L[1, j] v primeru, ko znaka nista enaka, max{L[0, j], L[1, j - 1]}. In ker je L[0, j] vedno 0, bo torej v primeru, ko znaka nista enaka, L[1, j] kar L[1, j - 1]. Če pa znaka sta enaka, velja L[1, j] = L[0, j - 1] + 1. In ker je L[0, *] vedno 0, bo torej tam, kjer imamo ujemanje y_j in x_0, L[1, j] vedno 1.

Torej lahko vrstico L[1, *] sestavimo tudi takole:

- Dokler y_j ni enak x_1, je L[1, j] = 0
- Od tistega j dalje (torej za jj > j) pa velja, da je L[1, jj] = 1

Poglejmo, kako se s tem "strinjajo" naše formule.

Pri j = 1 pogledamo znak na prvem mestu v nizu Y. Znak je enak znaku 'T'. Ker znaka nista enaka, uporabimo za izračun L[1, 1] formulo L[i, j] = max{L[i - 1, j], L[i, j - 1]}. To za naš podproblem pomeni L[1, 1] = max{L[0, 1], L[1, 0]} = max {0, 0} = 0.

Pri j = 2 pogledamo drugi znak v nizu Y. Znak je enak znaku 'T'. Znaka nista enaka, zato je L[1, 2] = max{L[0, 2], L[1, 1]} = max {0, 0} = 0.

Pri j = 3 pogledamo znak na tem (torej tretjem) mestu v nizu Y. Znak je enak znaku 'G'. Znaka sta enaka, zato za izračun L[1, 3] uporabimo formulo L[i, j] = L[i - 1, j - 1] +1. V našem zgledu to pomeni:
L[1, 3] = L[0, 2] +1 = 1.

Pri j = 4 pogledamo, ali se znak 'A' ujema z znakom 'G' iz niza X. Ker znaka nista enaka, L[1, 4] dobimo tako, da pogledamo vrednost pri L[0, 4] in vrednost pri L[1, 3] in vzamemo večjo vrednost.
L[1, 4] = max{L[0, 4], L[1, 3]} = max {0, 1} = 1.

Pri j = 5 pogledamo, ali se znak 'C' ujema z znakom 'G' iz niza X. Ker znaka nista enaka, L[1, 4] dobimo tako, da pogledamo vrednost pri L[0, 5] in vrednost pri L[1, 4] in vzamemo večjo vrednost.
L[1, 5] = max{L[0, 5], L[1, 4]} = max {0, 1} = 1.

Podobno je tudi pri j = 6. Kaj pa pri j = 7?
Pri j = 7 pogledamo sedmi znak v nizu Y. Tu je znak 'G', torej tak kot iz X. Znaka sta enaka, zato za izračun L[1, 7] uporabimo formulo L[i, j] = L[i - 1, j - 1] +1. V našem zgledu to pomeni:
L[1, 7] = L[0, 6] + 1 = 0 + 1 = 1.

Pri j = 8 je postopek tak, kot je bil pri j = 4, 5 in 6 in dobimo L[1, 8] = 1. S tem smo izračunali celotno prvo vrstico.

Korak 2:
Sedaj računamo drugo vrstico matrike L, torej L[2, j] za j = 1, …,8
V drugi vrstici iščemo dolžino skupnih podnizev niza X[1…2] in Y[1…j]. Niz X[1…2] je enak "GT". Iz prve vrstice vemo, kje smo našli skupno podzaporedje v X in Y, ki je enako "G". Sedaj ugotavljamo, ali je znak 'T' vsebovan tudi v nizu Y. Če dobro premislimo, ugotovimo, da če Y vsebuje 'T' in le-ta stoji pred znakom 'G', bo novo podzaporedje dolžine 1 in ne bo vsebovalo znaka 'G'. Če znak 'T' stoji po znaku 'G', pa bo novo podzaporedje dolžine 2 in bo enako "GT". Ker znak 'T' v nizu Y stoji le pred znakom 'G', zagotovo vemo, da podzaporedje ne more biti enako "GT". Ker najdemo znak 'T' v Y že na začetku, bo vrstica L[2, j] = 1.

Naredimo podoben razmislek kot na Koraku 1 in ugotovimo, da vrstico L[2, j] izračunamo na naslednji način:

- Dokler $x_2 \neq y_j$ in L[1, j] = 0, bodo vrednosti L[2, j] = 0.
- Če $x_2 \neq y_j$ in predhodno še nismo našli ujemanja, torej je L[2, j - 1] = 0, in je L[1, j] = 1, bodo vrednosti L[2, j] = 1.
- Če $x_2 = y_j$ in L[1, j] = 0, bodo od tistega j dalje, kjer smo našli ujemanje, vrednosti L[2, j] = 1.
- Če $x_2 = y_j$ in L[1, j - 1] = 1, bodo od tistega j dalje, kjer smo našli ujemanje, vrednosti L[2, j] = 2.

Izračunajmo nekaj členov s pomočjo formul:

Pri j = 1 pogledamo znak na tem mestu v nizu Y. Znak je enak znaku 'T'. Ker sta znaka enaka, uporabimo za izračun L[2, 1] formulo L[i, j] = L[i - 1, j - 1] + 1, kar pomeni L[2, 1] = L[1, 0] + 1 = 0 + 1 = 1.

Pri j = 2 pogledamo znak na tem mestu v nizu Y. Znak je enak znaku 'T'. Znaka sta enaka, zato je L[2, 2] = L[1, 1] + 1 = 0 + 1 = 1.

Pri j = 3 pogledamo znak na tem mestu v nizu Y. Znak je enak znaku 'G'. Znaka nista enaka, zato za izračun

L[2, 3] uporabimo formulo L[i, j] = max{L[i - 1, j], L[i, j - 1]}, kar za naš podproblem pomeni:
L[2, 3] = max{L[1, 3], L[2, 2]} = max{1, 1} = 1.

Postopek nadaljujemo, dokler ne pridemo do konca niza Y.

Korak 3:
Pri i = 3 in j = 1, …,8 računamo L[3, j].
V tretji vrstici iščemo dolžino skupnih podnizev niza X[1 … 3] in Y[1 … j]. Niz X[1 … 3] je enak "GTT". Iz prve in druge vrstice vemo, da niz Y vsebuje podzaporedja "G" in "T". Vemo, da podzaporedja "GT" ne vsebuje, torej tudi podzaporedja "GTT" zagotovo ne bomo našli v nizu Y. V nizu Y lahko najdemo kvečjemu podzaporedje, ki je enako "TT". Najprej pa moramo pogledati, če niz Y vsebuje tako podzaporedje, ki vsebuje znak 'T'.

Izračunajmo nekaj členov s pomočjo formul:
Izračunamo vrednost L[3, 1]. Znak na mestu j = 1 je enak znaku 'T', ki je enak znaku 'T' iz niza X. Zato za izračun vrednosti L[3, 1] uporabimo formulo L[i, j] = L[i - 1, j - 1] + 1. Torej bo
L[3, 1] = L[2, 0] + 1 = 0 + 1 = 1.

Pri j = 2, pogledamo znak na tem mestu v nizu Y. Znak je enak znaku 'T'. Znaka sta enaka, zato je
L[3, 2] = L[2, 1] + 1 = 1 + 1 = 2. Iz tega sledi, da je trenutno najdaljše skupno podzaporedje dolžine 2 in, da je enako "TT".

Znak na mestu j = 3, v nizu Y, je enak znaku 'G'. Ker znak 'G' ni enak znaku 'T', uporabimo za izračun vrednosti L[3, 3] formulo L[i, j] = max{L[i - 1, j], L[i, j - 1]}, kar za naš podproblem pomeni:
L[3, 3] = max{L[2, 3], L[3, 2]} = max{1, 2} = 2.

Postopek nadaljujemo, dokler ne pridemo do konca niza Y.

Podoben postopek ponavljamo, dokler je še kaj znakov v nizu X.

V našem primeru je dolžina niza X enaka 6, zato bo zadnji korak, ki ga bomo naredili, šesti. Oglejmo si ga.

Korak 6:
Na tem koraku bo i = 6 in j = 1, …, 8. Računamo L[6, j].
V zadnji vrstici iščemo dolžino skupnih podnizov X[1 … 6] in Y[1 … 8]. Do sedaj smo poiskali, izračunali dolžine najdenih podnizov za niz X[1 … 5] in Y[1 … 8]. Torej smo za vsa podzaporedja niza "GTTAC" preverili, ali niz Y vsebuje taka podzaporedja, ki so enaka podzaporedjem iz niza "GTTAC". V bistvu nas zanima, če bodo podzaporedja, ki smo jih že našli v nizu Y in jim dodamo še znak 'A', še vedno podzaporedja v nizu Y.

Iz splošne formule vemo, da bomo L[6, j] izračunali s pomočjo naslednjih formul:

$$L[\,6, j\,] = \begin{cases} L\,[\,5, j-1] + 1 & x_i = y_j \\ \max\,\{\,L\,[5, j], L[6, j-1]\} & x_i \neq y_j \end{cases}$$

Izračunajmo nekaj členov s pomočjo te formule:
Pri j = 1 pogledamo znak na tem mestu v nizu Y. Znak je enak znaku 'T'. Ker znaka nista enaka, uporabimo za izračun L[6, 1] formulo L[6, j] = max{L[5, j], L[6, j - 1]}, zato bo
L[6, 1] = max{L[5, 1], L[6, 0]} = max{1, 0} = 1.

Znak na mestu j = 2, v nizu Y, je 'T'. Znaka nista enaka, zato je
L[6, 2] = max{L[5, 2], L[6, 1]} = max{2, 1} = 2.

L[6, 3] = max{L[5, 3], L[6, 2]} = max{2, 2} = 2, zato, ker je na tretjem mestu v nizu Y znak enak 'G', ki pa ni enak znaku 'A' iz niza X.

Četrti znak v nizu Y je enak znaku 'A', ki je enak kot znak 'A' iz niza X. Zato velja
L[6, 4] = L[5, 3] + 1 = 2 + 1 = 3.

Ker je peti znak v nizu Y enak 'C' in ni enak znaku iz niza X, velja
L[6, 5] = max{L[5, 5], L[6, 4]} = max{4, 3} = 4.

Postopek nadaljujemo, dokler ne pridemo do konca niza Y.

Tako smo do konca zgradili matriko L:

			T	T	G	A	C	A	G	A
	L	0	1	2	3	4	5	6	7	8
	0	0	0	0	0	0	0	0	0	0
G	1	0	0	0	1	1	1	1	1	1
T	2	0	1	1	1	1	1	1	1	1
T	3	0	1	2	2	2	2	2	2	2
A	4	0	1	2	2	3	3	3	3	3
C	5	0	1	2	2	3	4	4	4	4
A	6	0	1	2	2	3	4	5	5	5

Sedaj lahko iz zgrajene matrike na mestu L[6, 8] razberemo, da je dolžina najdaljšega skupnega podzaporedja S nizov X in Y enaka 5.

Sestavljanje najdaljšega skupnega podzaporedja
Sestavili smo matriko L in vemo, kako dolgo je najdaljše skupno podzaporedje. Sedaj pa poglejmo še, kako s pomočjo matrike L to najdaljše skupno podzaporedje poiščemo.

X	G	T	T	A	C	A		
i	0	1	2	3	4	5		
Y	T	T	G	A	C	A	G	A
j	0	1	2	3	4	5	6	7

Korak 0:
Vzamemo zadnja znaka iz niza X in Y, znaka 'A' in 'A'. Ker sta znaka enaka, bo ta znak 'A' zagotovo del najdaljšega podniza. Zapomnimo si znak 'A'. V obeh nizih režemo zadnja znaka.

X	G	T	T	A	C		
i	0	1	2	3	4		
Y	T	T	G	A	C	A	G
j	0	1	2	3	4	5	6

Korak 1:
Sedaj je dolžina niza X enaka n = 5, Y pa m = 7. Primerjamo zadnja znaka 'C' in 'G'. Ker sta različna, pogledamo v matriko L. Vrednost L[5, 7] smo izračunali kot max{L[4, 7] , L[5, 6]} = max{3, 4}. Ker je L[5, 6] > L[4, 7], bomo zadnji znak rezali v nizu Y.

X	G	T	T	A	C	
i	0	1	2	3		
Y	T	T	G	A	C	A
j	0	1	2	3	4	5

Korak 2:
Znak 'C' ni enak znaku 'A'. Najdaljše skupno podzaporedje prvih pet znakov iz niza X in prvih šest znakov iz niza Y (L[5, 6]) smo izračunali kot max{L[4, 6] , L[5, 5]} = max{3, 4}. Ker je L[5, 5] > L[4, 6], bomo zadnji znak ponovno rezali v nizu Y.

X	G	T	T	A	C
i	0	1	2	3	
Y	T	T	G	A	C
j	0	1	2	3	4

Korak 3:
Ker sta zadnja znaka enaka, bo ta znak ('C') zagotovo del najdaljšega podniza. Zapomnimo si znak 'C'. V obeh nizih režemo zadnja znaka.

X	G	T	T	A
i	0	1	2	3
Y	T	T	G	A
j	0	1	2	3

Korak 4:
Primerjamo znaka 'A' in 'A'. Znaka sta enaka, zato bo 'A' del najdaljšega skupnega podniza. Znak si zapomnimo. V obeh nizih režemo zadnja znaka.

X	G	T	T
i	0	1	2
Y	T	T	G
j	0	1	2

Korak 5:
Primerjamo znaka 'T' in 'G'. Znaka sta različna. L[2, 2] smo izračunali z max{L[1, 2] , L[2, 1]}. Ker je L[1, 2] = 0 < L[2, 1] = 1, bomo zadnji znak rezali v nizu Y.

X	G	T	T
i	0	1	2
Y	T	T	
j	0	1	

Korak 6:
Primerjamo znaka 'T' in 'T'. Znaka sta enaka. Torej bo najdaljše skupno podzaporedje vsebovalo tudi znak 'T', zato si ga zapomnimo. V obeh nizih režemo zadnja dva znaka.

X	G	T
i	0	1
Y	T	
j	0	

Korak 7:
Zadnji znak iz niza Y je enak 'T'. Tudi zadnji znak iz niza X je enak 'T'. Torej bo znak 'T' del najdaljšega skupnega podzaporedja. Zapomnimo si znak in v obeh nizih režemo zadnja znaka.

X	G
i	0
Y	
j	

Ker niz Y ne vsebuje več znakov, vrnemo niz, ki ga dobimo tako, da iz zapomnjenih znakov sestavimo niz v obratni smeri, kot smo si znake zapomnili.
Iz tega sledi, da imata niza X = "GTTACA" in Y= "TTGACAGA" najdaljše skupno podzaporedje S enako "TTACA".

3.1.5 Algoritmi

V tem razdelku bomo opisali algoritma s katerima lahko rešimo problem iskanja najdaljšega skupnega podzaporedja, in sicer z naivnim pristopom in s pomočjo dinamičnega programiranja.

V razredu `LCS` bomo implemetirali metode `lcsNaivniPristop(String x, String y)`, `lcsDinamicno(String x, String y)` in `lcsDinamicnoMatrikaL(String x, String y)`. Poleg omenjenih metod bodo v tem razredu implementirane tudi privatne metode, ki jih bomo potrebovali. Metode bodo kot vhodna parametra dobile niza `x` in `y`, ki sta tipa `String`. Metodi `lcsNaivniPristop(String x, String y)`, `lcsDinamicno(String x, String y)` bosta kot rezultat vrnili niz `s`, ki je tudi tipa `String` in predstavlja najdaljše skupno podzaporedje, ki je hkrati podzaporedje niza `x` in `y`. Metoda `lcsDinamicnoMatrikaL(String x, String y)` bo vrnila matriko `L`, v kateri so podatki tipa `int`.

```
import java.util.*;
public class LCS
{

  public static String lcsNaivniPristop(String x, String y)
  {
    //implementacija rešitve LCS problema s pomočjo naivnega pristopa
  }

  public static String lcsDinamicno(String x, String y)
  {
    //implementacija rešitve LCS problema s pomočjo metode dinamičnega
   //programiranja
  }

  public static int[][] lcsDinamicnoMatrikaL(String x, String y)
  {
    //implementacija računanja matrike L
  }

}
```

Za prikaz uporabe algoritmov bomo napisali poseben razred `LCSUporaba`, ki bo v metodi `main` klical zgoraj omenjene algoritme. V tej metodi bomo predpostavili, da imamo niz `X` shranjen v spremenljivki `x` tipa `String`, niz `Y` pa v spremenljivki `y`, ki je tudi tipa `String`. Iščemo najdaljše skupno podzaporedje, ki je hkrati tudi podzaporedje niza `x` in `y`. Oglejmo si program za prikaz uporabe algoritmov, zapisan v javanski kodi.

```
import java.util.*;
public class LCSUporaba
{
  public static void main(String [ ] args)
  {
    String x = "TTGACAGA"; //dani niz X
    String y = "GTTACA"; //dani niz Y

    //pokličemo metodo, ki izpiše iskano podzaporedje S, s pomočjo
    //metode naivnega pristopa
    System.out.print("Naivni pristop: ");
    lcsNaivniPristopUporaba(x, y);

    //pokličemo metodo, ki izpiše iskano podzaporedje S, s pomočjo
    //metode dinamičnega programiranja
    System.out.print("Metoda dinamičnega programiranja: ");
    lcsDinamicnoUporaba(x, y);

    //pokličemo metodo, ki izpiše matriko L
    System.out.println("Izpis matrike L: ");
    lcsDinamicnoIzpisLUporaba(x, y);
  }

  private static void lcsNaivniPristopUporaba(String x, String y)
  {
    //pokličemo algoritem LCS, ki nam vrne najdaljše skupno
    //podzaporedje niza x in y
    //s pomočjo metode naivnega pristopa
    String najdaljsePodzaporedje = LCS.lcsNaivniPristop(x, y);

    //če je spremenljivka najdaljsePodzaporedje enaka praznemu nizu,
    //najdaljšega skupnega podzaporedja nismo našli
    if(najdaljsePodzaporedje != "")
    {
      System.out.println(najdaljsePodzaporedje);
    }
    else
    {
      System.out.println("Najdaljšega skupnega podzaporedja nismo
          našli.");
    }
  }

  private static void lcsDinamicnoUporaba(String x, String y)
  {
    //pokličemo algoritem LCS, ki nam vrne najdaljše skupno
    //podzaporedje niza x in y
    //s pomočjo metode dinamičnega programiranja
    String najdaljsePodzaporedje = LCS.lcsDinamicno(x, y);

    //če je spremenljivka najdaljsePodzaporedje enaka praznemu nizu,
    //najdaljšega skupnega podzaporedja nismo našli
    if(najdaljsePodzaporedje != "")
    {
      System.out.println(najdaljsePodzaporedje);
    }
    else
    {
      System.out.println("Najdaljšega skupnega podzaporedja nismo
          našli.");
    }
  }
}
```

```
  private static void lcsDinamicnoIzpisLUporaba(String x, String y)
  {
    //pokličemo LCS algoritem, ki nam vrne matriko L
    int[][] l = LCS.lcsDinamicnoMatrikaL(x, y);

    //izpis L matrike
    int stVrstic = l.length; //število vrstic v matriki
    int stStolpcev = l[0].length; //število stolpcev v matriki
    for (int i = 0; i < stVrstic; i++)
    {
      for (int j = 0; j < stStolpcev; j++)
      {
        System.out.print(l[i][j]);
        //pogledamo, če trenutni element je zadnji element v vrstici
        if (j < stStolpcev - 1)
        {
          //če ni zadnji element izpišemo presledek
          System.out.print(" ");
        }
      }
      //konec vrstice, gremo v novo vrstico
      System.out.println();
    }
  }

}
```

3.1.5.1 Naivni pristop

Kot že vemo, je dani problem naslednji: podana imamo niza `X` in `Y`. Iščemo najdaljše skupno podzaporedje, ki je hkrati podzaporedje niza `X` in `Y`. Tako sta na primer za dana niza `X = "abc"` in `Y = "acb"` dve taki podzaporedji. To sta "ab" in "ac".
Naivni pristop reševanja LCS problema je, da pregledamo vsa podzaporedja v X in vzamemo najdaljše podzaporedje, ki je tudi v Y.
Najprej sestavimo vsa možna podzaporedja niza `X`. To so: `a, ab, abc, ac, b, bc, c`. V ta namen najprej napišimo rekurzivno privatno metodo `vsaPodzaporedjaNizaX(String x, Stack<String> podzaporedjax)`, ki nam bo pomagala poiskati vsa možna podzaporedja niza `x`. Metoda bo kot vhodna parametra vzela niz `x` tipa `String`, in referenco na sklad `podzaporedjaX`, v katerem bomo hranili podatke, podzaporedja niza `x`, ki bodo tipa `String`.

```
private static void vsaPodzaporedjaNizaX(String x,
        Stack<String> podzaporedjax)
{
  //vzamemo prvo podzaporedje, ki je na vrhu sklada
  //s pomočjo metode peek(), le preberemo vrhnji element iz sklada
  String podniz = podzaporedjax.peek();

  //dokler je še kaj znakov v nizu x
  for (int i = 0; i < x.length(); i++)
  {
     //v sklad vpišemo najdeno podzaporedje
     podzaporedjax.push(podniz + x.charAt(i));

    //rekurzivno kličemo metodo
    //režemo črke v nizu x
    vsaPodzaporedjaNizaX(x.substring(i + 1), podzaporedjax);
  }
}
```

Sedaj pa napišimo še metodo `vrniPodzaporedja(String x)`. Ta bo klicala metodo `vsaPodzaporedjaNizaX(String x, Stack<String> podzaporedjax)`. Vrnila bo sklad, v kateremu bodo vsa podzaporedja niza `x`. Metoda bo javna, da jo lahko uporabimo tudi v razredu `LCSUporaba`.

```
public static Stack<String> vrniPodzaporedja(String x)
{
  //naredimo nov objekt tipa Stack (sklad)
  Stack<String> podz = new Stack<String>();
  //v sklad podz damo prazen niz
  podz.push("");
  //poiščemo vsa podraporedja v nizu x
  NajdiVsaPodzaporedjaNizaX(x, podz);
  //vrnemo vsa podzaporedja niza x
  return podz;
}
```

Podzaporedja niza `x` smo shranjevali v podatkovno strukturo sklad zato, da se ne obremenjujemo s tem, ali smo trenutno podzaporedje iz niza `x` že iskali v nizu `y`. Objekt sklad oziroma `Stack` vsebuje metodo `pop()`, ki vrne zgornji element in ga izbriše iz sklada. Torej dokler bo še kaj elementov v skladu, bomo vzeli zgornji element iz sklada in ga shranili v spremenljivko `podzx`, ki bo tipa `String`. Nato bomo pogledali, ali `podzx` obstaja v nizu `y`. To bomo storili na naslednji način:
Recimo, da je podzaporedje niza `x` enako `"ab"` in ga označimo z `podzx1`. Niz `y` je enak `"acb"`. Vzeli bomo prvi znak iz niza `x` in pogledali, ali obstaja tudi v nizu `y`. Torej vzeli bomo znak `'a'` iz `podzx1` in pogledali, če je enak prvemu znaku iz niza `y`, znaku `'a'`. Ker sta znaka enaka, se bomo pomaknili za en znak v desno v `podzx1` in v nizu `y`. Na drugem koraku bomo primerjali, ali je znak `'b'` iz `podzx1` enak znaku `'c'` iz `y`. Ker znaka nista enaka, se bomo v nizu `y` premaknili za en znak v desno. Na tretjem koraku bomo primerjali znak `'b'` iz `podzx1` z znakom `'b'` iz `y`. Znaka sta enaka. Ker `podzx1` ne vsebuje več znakov za primerjanje, s primerjanji zaključimo. V tem primeru je niz `"ab"` eno izmed skupnih podzaporedij niza `x` in `y` in hkrati tudi najdaljše.

```
public static String lcsNaivniPristop(String x, String y)
{
  int m = y.length(); //dolžina niza y
  //pokličemo metodo, ki poišče vsa podzaporedja niza x
  //in jih shrani v sklad podzaporedjax
  Stack<String> podzaporedjax = vrniPodzaporedja(x);
  //v spremenljivko najdaljsi bomo shranili najdaljši
  //skupni podniz niza x in y
  String najdaljsi = "";
  //dokler sklad podzaporedjax ni prazen
  while(!podzaporedjax.empty())
  {
    //vzamemo prvo podzaporedje niza x iz sklada podzaporedjax
    String podzx = String.valueOf(podzaporedjax.pop());

    //v metodi vrniPodzaporedja smo naredili nov sklad in mu dodali
    //prazen niz
    //to pomeni, da je prvo podzaporedje niza x enako praznemu nizu
    //in je shranjeno na koncu sklada podzaporedjax
    //če je podzx enako praznemu nizu, končamo s preverjanji
    if (podzx == "")
      break;

    //v spremenljivko skupnipodniz bomo shranjevali najdeni skupni
    //podniz niza x in y
    //za vsako novo preverjanje se bo skupnipodniz nastavil na
    //prazen niz
    String skupnipodniz = "";

    //s pomočjo spremenljivke j se bomo sprehajali po nizu y
    int j = 0;
    //s pomočjo spremenljivke k se bomo sprehajali po nizu podzx
    int k = 0;

    //dokler ne pridemo do konca niza y, ponavljamo
    while(j < m)
    {
      //pogledamo če je k-ti znak iz niza podzx enak j-temu znaku
      //niza y
      if(podzx.charAt(k) == y.charAt(j))
      {
        //k-ti in j-ti znak sta enaka
        //shranimo znak v spremenljivko skupnipodniz
        skupnipodniz += y.charAt(j);

        //povečamo k za ena -> premaknemo se za en znak v desno v
        //nizu podzx
        k = k + 1;

        //če v nizu podzx ni več znakov za primerjat, končamo s
        //primerjanji
        if(k >= podzx.length())
        {
          break;
        }
        else
        {
          //sicer povečamo j za ena -> premaknemo se za en znak v
          //desno v nizu y
          j = j + 1;
        }
      }
```

```
      else //k-ti in j-ti znak nista enaka
      {
        //iščemo naprej ali je k-ti znak vsebovan v nizu y
        //povečamo j za ena -> premaknemo se za en znak v desno v
        //nizu y
        j = j + 1;
      }
    }

    //pogledmo dolžino trenutno najdaljšega najdenega skupnega
    //podzaporedja
    //če je enaka dolžini niza x, pomeni, da je celoten niz x
    //vsebovan v nizu y
    //zato je vrednost v spremenljivki skupnipodniz zagotovo
    //najdaljse skupno podzaporedje
    if(skupnipodniz.length() == x.length())
      return skupnipodniz; //vrnemo najdaljse skupno podzaporedje

    //sicer moramo najdaljše podzaporedje še iskati
    //pogledamo ali je dolžina trenutno najdenega skupnega
    //podzaporedja niza x in y večja od dolžine trenutno najdaljšega
    //najdenega skupnega podzaporedja
    //če je večja, niz skupnipodniz shranimo kot trenutno najdaljše
    //skupno podzaporedje niza x in y
    if(skupnipodniz.length() > najdaljsi.length())
    {
      najdaljsi = skupnipodniz;
    }
  }

  //vrnemo najdalše skupno podzaporedje, ki je hkrati podzaporedje
  //niza x in y
  return najdaljsi;
}
```

Naivni pristop lahko nekoliko izboljšamo, če podnize gradimo tako, da so v skladu urejeni. To pomeni, da je na vrhu sklada najdaljši podniz, na dnu sklada pa najkrajši podniz. Če bi imeli sklad urejen na omenjeni način, nam ne bi bilo potrebno pregledovati, če je trenutno najdeno skupno podzaporedje v nizu X in Y najdaljše. Vedeli bi, da je najdaljše, saj bi bili podnizi v skladu tako urejeni.

3.1.5.2 Pristop s pomočjo metode dinamičnega programiranja

V tem razdelku bomo opisali algoritem za iskanje najdaljšega skupnega podzaporedja, ki je hkrati podzaporedje nizov `x` in `y` s pomočjo metode dinamičnega programiranja. Napisali bomo dve metodi, in sicer `lcsDinamicnoIzpisMatrike(String x, String y)` in `lcsDinamicnoProgramiranje(String x, String y)`. Prva metoda bo izračunala in vrnila matriko `L`, druga pa nam bo s pomočjo matrike `L` in rekurzije vrnila najdaljše skupno podzaporedje.

Algoritem `lcsDinamicnoIzpisMatrike` je zelo enostaven. Vhodna podatka sta niz `x` in `y`, ki sta tipa `String`. Izhodni podatek je matrika `L` tipa `int`. Iz matrike lahko razberemo dolžino `L[i, j]` najdaljšega podzaporedja S, ki je hkrati podzaporedje niza `x[0, …,i]` = $x_0\ x_1\ \dots\ x_i$ in `y[0, …,j]` = $y_1\ y_2\ \dots\ y_j$.

Na začetku definiramo matriko velikosti `(n + 1) x (m + 1)`, kjer je `n` dolžina danega niza `x`, `m` pa dolžina danega niza `y`. Ko je `j = 0`, bodo vrednosti `L[i, 0]`, kjer je `i = 0, …,n`, enake `0`. Ko je `i = 0`, bodo vrednosti `L[0, j]`, kjer je `j = 0, …,m`, enake `0`. Nato začnemo z gradnjo matrike in upoštevamo pogoje:

Če je `i`-ti znak v nizu `x` enak `j`-temu znaku v nizu `y`, upoštevamo formulo

```
L[i, j] = L[i - 1, j - 1] + 1,
```

sicer upoštevamo formulo

```
L[i, j] = max{L[i - 1, j], L[i, j - 1]}.
```

```
public static int[][] lcsDinamicnoIzpisMatrike(String x, String y)
{
  int n = x.length(); //dolžina niza x
  int m = y.length(); //dolžina niza y

  //definiramo matriko velikosti(n + 1) x (m + 1)
  int[][] l = new int[n+1][m+1];

  //ko je j=0, bodo vrednosti v prvem stolpcu matrike enake 0
  for(int i = 0; i <= n; i++)
  {
    l[i][0] = 0;
  }

  //ko je i = 0, bodo vrednosti v prvi vrstici matrike enake 0
  for(int j = 0; j <= m; j++)
  {
    l[0][ j] = 0;
  }

  //začnemo z gradnjo matrike
  for(int i = 1; i <= n; i++)
  {
    for(int j = 1; j <= m; j++)
    {
      //pogledamo, če sta znaka enaka
      if(x.charAt(i-1) == y.charAt(j-1))
      {
        l[i][j] = l[i-1][j-1] +1; //znaka sta enaka
      }
      else
        l[i][j] = Math.max(l[i-1][j], l[i][j-1]);//znaka nista enaka
    }
  }
  return l; //vrnemo matriko celih števil
}
```

Sedaj uporabimo zgrajeno matriko `L`, ki nam pove dolžino najdaljšega podzaporedja niza `x` in `y`, da ugotovimo katero podzaporedje je to.

Napisali bomo metodo `lcsDinamicnoProgramiranje(String x, String y)`, ki bo kot vhodna parametra vzela niza `x` in `y`, ki sta tipa `String`. Kot izhodni podatek bo vrnila najdaljše skupno podzaporedje, ki je hkrati podzaporedje niza `x` in `y`. Tudi rezultat bo seveda tipa `String`. Tu bomo preverjanje, ali sta znaka enaka, začeli od konca niza `x` in `y` in se premikali proti začetku. Torej, če dolžino niza `x` označimo z `n` in dolžino niza `y` označimo z `m`, bomo na prvem koraku preverili, ali sta zadnja znaka enaka, torej ali je znak na mestu `x[n - 1]` enak znaku na mestu `y[m - 1]`. Če sta znaka enaka, se bomo premaknili za ena v levo po obeh nizih. Sicer bomo pogledali v matriko `L`, ki nam bo povedala, po katerem nizu se premaknemo v levo. V matrki `L` pogledamo vrednost na mestu `L[n][m - 1]` in vrednost `L[n - 1][m]`. Če je prva vrednost manjša od druge, se premaknemo za ena v levo po nizu `x`, sicer se premaknemo za ena v levo po nizu `y`.

```
public static String lcsDinamicno(String x, String y)
{
  //pokličemo metodo, ki nam vrne matriko L
  int[][] L = lcsDinamicnoIzpisMatrike(x, y);
  int n = x.length();//dolžina niza x
  int m = y.length();//dolžina niza y
  //najdaljše skupno podzaporedje niza x in y bomo shranili v
  //spremenljivko najdaljse
  String najdaljse = "";

  //če je eden od nizov prazen, skupnega podzaporedja ne moremo
  //iskat
  if(x == null || y == null || n == 0 || m == 0)
    return "";

  //pogledamo če sta znaka enaka
  if(x.charAt(n-1) == y.charAt(m-1)) //znaka sta enaka
  {
    //rekurzivno kličemo metodo
    najdaljse = LCSDinamicno(x.substring(0, n-1), y.substring(
      0, m-1)) + String.valueOf(x.charAt(n-1));
    return najdaljse;
  }
  else //znaka nista enaka
  {
    //pogledamo vrednosti v matriki
    if(L[n][m-1] < L[n-1][m])
      najdaljse = LCSDinamicno(x.substring(0, n-1), y);//zmanjšamo x
    else if(L[n][m-1] => L[n-1][m])
      najdaljse = LCSDinamicno(x, y.substring(0, m-1));//zmanjšamo y
  }

  //vrnemo najdaljše skupno podzaporedje, ki je hkrati podzaporedje
  //niza x in y
  return najdaljse;
}
```

3.1.6 Časovna zahtevnost

3.1.6.1 Naivni pristop

Oglejmo si časovno zahtevnost algoritma LCS, če ga rešujemo z naivnim pristopom.
Velikost problema, ki ga obravnavamo, je odvisna od dolžine obeh nizov, ki sta vhodna parametra. Kot vhodna parametra algoritem sprejme niza X in Y. Dolžini X, ki jo označimo z n in Y, ki jo označimo z m, predstavljata velikost problema. Za algoritem LCS, ki ga rešujemo z naivnim pristopom in ga označimo z np, določimo časovno zahtevnost problema $T_{np}(n, m)$.

Algoritem najprej poišče vsa možna podzaporedja v nizu X. Če dolžino niza X označimo z n, imamo 2^n različnih podzaporedij. Nato za vsako podzaporedje v nizu X pogledamo, ali obstaja tudi v nizu Y. Označimo dolžino niza Y z m. Torej se za vsako podzaporedje niza X sprehodimo po nizu Y. Tako naredimo približno $2^n * m$ primerjanj.
Časovna zahtevnost algoritma LCS, ki ga rešujemo z naivnim pristopom, je torej enaka $O(2^n * m)$.

Oglejmo si, kakšna je časovna zahtevnost algoritma v najboljšem, pričakovanem in najslabšem primeru.

Najboljši primer

Zanemarimo očitno nesmiselen primer, ko je eden od nizov prazen. Če dobro premislimo, opazimo, da je tudi v najboljšem primeru časovna zahtevnost odvisna od dolžine niza X in Y. V vsakem primeru iščemo vsa podzaporedja v nizu X in nato iščemo, ali obstajajo tudi v nizu Y. Zraven si še beležimo trenutno najdaljše podzaporedje v nizu X, ki je hkrati v nizu Y. Najboljši primer bo torej nastopil, ko bo Y cel na začetku niza X. Torej sta niza na primer oblike X = "anja" in Y = "anj" ali X = "zemljevid" in Y = "zemlj". Ta podatek nam ne koristi, saj moramo v vsakem primeru pregledati vsa podzaporedja iz niza X in jih primerjati s podzaporedji iz niza Y. Torej je dejansko časovna zahtevnost odvisna od dolžine nizov

$$T_{np}(n, m) = O(2^n * m)$$

Najslabši primer

Najslabši primer nastopi, ko niza x in y ne vsebujeta skupnega podzaporedja. Torej se niti eden znak iz niza X ne pojavi v nizu Y, denimo "abcdefghijklmn" in "ooooooooo". Tedaj smo primerjanja izvajali po nepotrebnem.

$$T_{np}(n, m) = O(2^n * m)$$

Pričakovana časovna zahtevnost

Ker je analiza pričakovane časovne zahtevnosti zapletena, povejmo le, da je enaka $O(2^n * m)$.

3.1.6.2 Pristop s pomočjo metode dinamičnega programiranja

Oglejmo si časovno zahtevnost LCS algoritma, če ga rešujemo s pomočjo metode dinamičnega programiranja.

Velikost problema, ki ga obravnavamo, je odvisna od dolžine obeh nizov, ki sta vhodna parametra. Kot vhodna parametra algoritem sprejme niza X in Y. Dolžini X, ki jo označimo z n in Y, ki jo označimo z m, predstavljata velikost problema. Za LCS algoritem, ki ga rešujemo z metodo dinamičnega programiranja in ga označimo z dp, določimo časovno zahtevnost problema $T_{dp}(n, m)$.

Časovna zahtevnost algoritma je odvisna od gradnje matrike L. Matriko gradimo s pomočjo dveh zank. S prvo zanko se sprehajamo po vrsticah, z drugo pa po stolpcih. Vrstic je toliko, kolikor je dolžina niza X, stolpcev pa toliko kot je dolžina niza Y. Znotraj zank opravimo konstantno mnogo dela. Torej je časovna zahtevnost algoritma LCS enaka $O(n * m)$.

$$T_{dp}(n, m) = O(n * m)$$

Literatura in viri

Boštjan Jaklič, 2004/2005 : Seminarska naloga Huffmanovo kodiranje (online). Dostopno na naslovu:
http://www.educa.fmf.uni-lj.si/www375/2005_6/PSIA2005_6/seminarska/Huffmanovo%20kodiranje/Jaklic/Huffmanovo_kodiranje.htm
(zadnji obisk: 14.4.2009)

Department of Computer Science, Kent State University : Boyer-Moore Algorithm (online). Dostopno na naslovu:
http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/StringMatch/boyerMoore.htm
(zadnji obisk: 14.4.2009)

Goodrich, M. T., Tamassia, R., 2001: *Data structures and algorithms in java*, J.Wiley & Sons : New York, 543 str.

MaFiRaWiki, 2007 : Huffmanovo kodiranje (online). Dostopno na naslovu:
http://wiki.fmf.uni-lj.si/wiki/Huffmanovo_kodiranje
(zadnji obisk: 14.4.2009)

R2Wiki, 2007 : Huffmanovo kodiranje (online). Dostopno na naslovu:
http://penelope.fmf.uni-lj.si/r2wiki/index.php/Huffmanovo_kodiranje
(zadnji obisk: 14.4.2009)

Sun Microsystems, Inc., 1995-2008 : The Java Tutorials (online). Dostopno na naslovu:
http://java.sun.com/docs/books/tutorial/
(zadnji obisk: 14.4.2009)

Vilfan, B., 2002 : *Osnovni algoritmi*, Ljubljana : Fakulteta za računalništvo in informatiko, 177 str.

Warwick, Department of Computer Science, 2008/2009 : CS301 Complexity of algorithms, PART 2, lessons 1, 2, 3, 4, 5, 7 (online). Dostopno na naslovu:
http://www2.warwick.ac.uk/fac/sci/dcs/teaching/material/cs301/
(zadnji obisk: 14.4.2009)

Wikimedia Foundation, Inc., 2007 : Boyer–Moore string search algorithm (online). Dostopno na naslovu:
http://en.wikipedia.org/wiki/Boyer%E2%80%93Moore_string_search_algorithm
(zadnji obisk: 14.4.2009)

Wikimedia Foundation, Inc., 2009 : Longest common subsequence problem (online). Dostopno na naslovu:
http://en.wikipedia.org/wiki/Longest_common_subsequence_problem
(zadnji obisk: 14.4.2009)

The Gaspard-Monge, Institute of electronics and computer science, 1997 : Brute force algorithm (online). Dostopno na naslovu:
http://www-igm.univ-mlv.fr/~lecroq/string/node3.html
(zadnji obisk: 14.4.2009)