

Algoritmusok és adatszerkezetek II.
előadásjegyzet:

Mintaillesztés, Tömörítés

Ásványi Tibor – asvanyi@inf.elte.hu

2020. december 1.

Tartalomjegyzék

1. Mintaillesztés ([2] 32; [4])	4
1.1. Egyszerű mintaillesztő (brute-force) algoritmus	4
1.2. Quicksearch	5
1.3. Mintaillesztés lineáris időben (Knuth-Morris-Pratt, azaz KMP algoritmus)	7
2. Információtömörítés ([5] 5; [4])	14
2.1. Naiv módszer	14
2.2. Huffman-kód	14
2.2.1. Huffman-kódolás szemléltetése	16
2.3. Lempel–Ziv–Welch (LZW) módszer	18

Hivatkozások

- [1] ÁSVÁNYI TIBOR, Algoritmusok és adatszerkezetek II.
Útmutatások a tanuláshoz, jelölések, tematika,
fák, gráfok,
mintaillesztés, tömörítés
<http://aszt.inf.elte.hu/~asvanyi/ad/ad2jegyzet/>
- [2] CORMEN, T.H., LEISERSON, C.E., RIVEST, R.L., STEIN, C.,
magyarul: Új Algoritmusok, *Scolar Kiadó*, Budapest, 2003.
ISBN 963 9193 90 9
angolul: Introduction to Algorithms (Third Edititon),
The MIT Press, 2009.
- [3] FEKETE ISTVÁN, Algoritmusok jegyzet
<http://ifekete.web.elte.hu/>
- [4] KORUHELY GÁBOR, SZALAY RICHÁRD,
Algoritmusok és adatszerkezetek 2, 2015/16 tavaszi félév
(hallgatói jegyzet, lektorált és javított)
<http://aszt.inf.elte.hu/~asvanyi/ad/>
- [5] RÓNYAI LAJOS – IVANYOS GÁBOR – SZABÓ RÉKA, Algoritmusok,
TypoTeX Kiadó, 1999. ISBN 963 9132 16 0
https://www.tankonyvtar.hu/hu/tartalom/tamop425/2011-0001-526_ronyai_algoritmusok/adatok.html
- [6] WEISS, MARK ALLEN, Data Structures and Algorithm Analysis,
Addison-Wesley, 1995, 1997, 2007, 2012, 2013.
- [7] ÁSVÁNYI TIBOR, Algoritmusok és adatszerkezetek I. előadásjegyzet
(2019)
<http://aszt.inf.elte.hu/~asvanyi/ad/ad1jegyzet.pdf>

1. Mintaillesztés ([2] 32; [4])

Adott a $\Sigma = \{\sigma_1, \sigma_2, \dots, \sigma_d\}$ ábécé. ($1 \leq d < \infty$ konstans).

A $T/1 : \Sigma[n]$ szövegben keressük a $P/1 : \Sigma[m]$ minta előfordulásait ($1 \leq m \leq n$). A fenti szimbólumokat az egész fejezetben így fogjuk használni.

1.1. Definíció. $s \in 0..(n-m)$ pontosan akkor a P érvényes eltolása T -n, ha $T[s+1..s+m] = P[1..m]$.

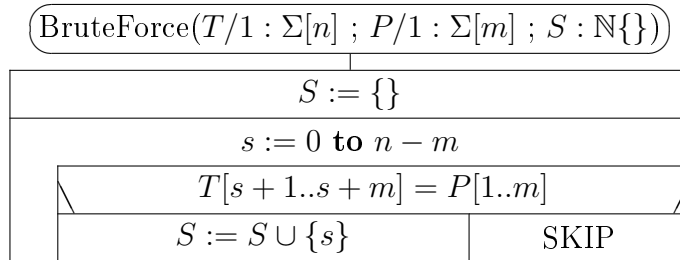
Az érvényes eltolások halmazát szeretnénk meghatározni, azaz az $S = \{s \in 0..(n-m) \mid T[s+1..s+m] = P[1..m]\}$ halmazt.

1.1. Egyszerű mintaillesztő (brute-force) algoritmus

Vegyük bevezetésként a következő példát! A $P[1..4] = BABA$ mintát keressük a $T[1..11] = ABABBABABAB$ szövegben. (B: B -t sikeresen illesztette a szöveg megfelelő betűjére; \cancel{B} : sikertelenül illesztette.)

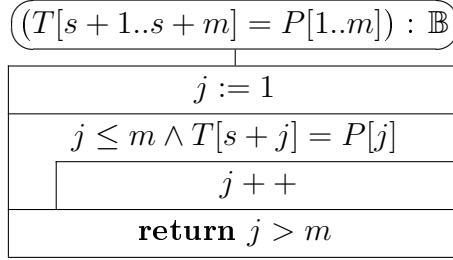
$i =$	1	2	3	4	5	6	7	8	9	10	11
$T[i]=$	A	B	A	B	B	A	B	A	B	A	B
	\cancel{B}	A	B	A							
		<u>B</u>	<u>A</u>	<u>B</u>	\cancel{A}						
			\cancel{B}	A	B	A					
				<u>B</u>	\cancel{A}	B	A				
$s=4$					<u>B</u>	<u>A</u>	<u>B</u>	<u>A</u>			
						\cancel{B}	A	B	A		
$s=6$							<u>B</u>	<u>A</u>	<u>B</u>	<u>A</u>	
								\cancel{B}	A	B	A

$$S = \{4; 6\}$$



A $T[s+1..s+m] = P[1..m]$ egyenlőségvizsgálatra: $MT_e(m) \in \Theta(m)$
 $mT_e(m) \in \Theta(1)$

Innét a teljes algoritmusra: $MT_{BF}(n, m) \in \Theta((n - m + 1) * m)$
 $mT_{BF}(n, m) \in \Theta(n - m + 1)$



Ha feltehető, hogy valamely $k < 1$ konstansra $m \leq k * n$, akkor $1 * n \geq n - m + 1 > n - k * n = (1 - k) * n$, ahol $1 - k > 0$ konstans. Innét $n - m + 1 \in \Theta(n)$ és $(n - m + 1) * m \in \Theta(n * m)$. Ebből pedig az $\cdot \in \Theta(\cdot)$ reláció tranzitivitása miatt adódik:

$$k < 1 \wedge m \leq k * n \Rightarrow mT_{\text{BF}}(n, m) \in \Theta(n) \wedge MT_{\text{BF}}(n, m) \in \Theta(n * m)$$

Ez viszont azt jelenti, hogy amennyiben m az n -hez képest nem is elhanyagolható, azaz valamely $\varepsilon > 0$ konstansra $m \geq \varepsilon * n$, akkor $\varepsilon * n * n \leq n * m \leq n * n$. Következésképp $n * m \in \Theta(n^2)$. A fentieket összegezve, ha az ε és a k konstansokra

$$\varepsilon > 0 \wedge k < 1 \wedge \varepsilon * n \leq m \leq k * n \Rightarrow MT_{\text{BF}}(n, m) \in \Theta(n^2)$$

1.2. Quicksearch

A gyorsabb keresés érdekében ennél és a következő (KMP) algoritmusnál általában egynél nagyobb lépésekben növeljük a $P[1..m]$ minta eltolását a $T[1..n]$ szöveghez képest úgy, hogy biztosan ne ugorjunk át egyetlen érvényes eltolást sem. Mindkét algoritmus a tényleges mintaillesztés előtt egy előkészítő fázist hajt végre, ami nem függ a szövegtől, csak a mintától.

A Quicksearch-nél ebben az előkészítő fázisban az ábécé σ elemeihez $shift(\sigma) \in 1..m+1$ címkéket társítunk, ahol $P[1..m]$ a keresett minta.

Tegyük fel most, hogy $\sigma = T[s + m + 1]$. Ekkor a $shift(\sigma)$ érték megmondja, hogy a $T[s + 1..s + m] = P[1..m]$ összehasonlítás után legalább mennyivel kell (jobbra) eltolni a P mintát a szövegen ahhoz, hogy a $T[s + m + 1]$ alapján legyen esély a mintának a megfelelő szövegrészhez való illeszkedésére.

– $\sigma \in P[1..m]$ esetén a $shift(\sigma) \in 1..m$ érték azt mondja meg, hogy legalább mennyivel kell tovább tolni a P mintát ahhoz, hogy a $T[s + m + 1]$ betűhöz kerülő karaktere maga is σ legyen. Világos, hogy a σ legjobboldali P -beli előfordulásához tartozik a legkisebb ilyen eltolás.

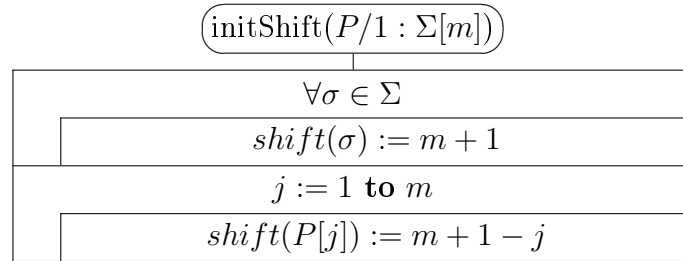
– $\sigma \notin P[1..m]$ esetén $shift(\sigma) = m + 1$ lesz, azaz a minta *átugorja* a $T[s + m + 1]$ karaktert.

Arra az esetre, amikor az ábécé $\Sigma = \{A,B,C,D\}$, a minta pedig $P[1..4]=CADA$, az alábbi félig absztrakt példákban xxxx mutatja a CADA mintával az eltolás előtt összehasonlított szövegrészt, maga a CADA pedig a minta eltolás utáni helyzetét. (Ezután természetesen újabb összehasonlítás kezdődik a szöveg megfelelő része és a minta között stb.)

Szöveg: ...xxxxA.....xxxxB.....xxxxC.....xxxxD...
Minta: CADA CADA CADA CADA

A megfelelő *shift* értékek értékeit a következő táblázat mutatja.

σ	A	B	C	D
$shift(\sigma)$	1	5	4	2



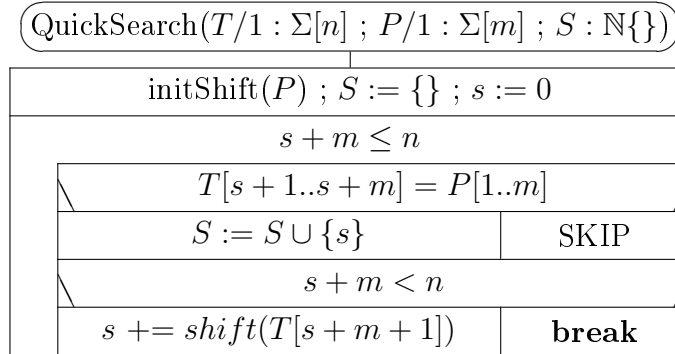
Az ábécé méretét konstansnak tekintve $T_{\text{initShift}}(m) \in \Theta(m)$ adódik.

A $P[1..4] = CADA$ mintával az $\text{initShift}()$ majd a Quicksearch működése:

σ	A	B	C	D
initial $shift(\sigma)$	5	5	5	5
C			4	
A	3			
D			2	
A	1			
final $shift(\sigma)$	1	5	4	2

$i =$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
$T[i]=$	A	D	A	B	A	B	C	A	D	A	B	C	A	B	A	D	A	C	A	D	A	D	A
	\emptyset	A	D	A																			
		\emptyset	A	D	A																		
$s = 6$							<u>C</u>	<u>A</u>	<u>D</u>	<u>A</u>													
												<u>C</u>	<u>A</u>	\emptyset	A								
														\emptyset	A	D	A						
$s = 17$																	<u>C</u>	<u>A</u>	<u>D</u>	<u>A</u>			
																			\emptyset	A	D	A	

$$S = \{6; 17\}$$



$mT(n, m) \in \Theta\left(\frac{n}{m+1} + m\right)$ (pl. ha $T[1..n]$ és $P[1..m]$ diszjunktak)
 $MT(n, m) \in \Theta((n - m + 2) * m)$ (pl. ha $T = AA \dots A$ és $P = A \dots A$)

A minimális műveletigény nagyságrenddel jobb, mint az egyszerű mintaillesztésnél, a tényleges (nem az aszimptotikus) maximális futási idő viszont még egy kicsit rosszabb is. Szerencsére, a tapasztalatok szerint az átlagos futási idő inkább a legjobb esethez áll közel, mintsem a legrosszabbhoz. Időkritikus alkalmazásokban viszont egy stabilabb futási idejű, minden esetben hatékony algoritmusra lenne szükségünk.

1.3. Mintaillesztés lineáris időben (Knuth-Morris-Pratt, azaz KMP algoritmus)

Tekintsük bevezetésként a következő példát! A $P[1..8] = BABABBAB$ mintát keressük a $T[1..18] = ABABABABBABABABBAB$ szövegben. (A minta elején a jelöletlen betűkről „illesztés nélkül is tudja” az algoritmus, hogy illeszkednek a szöveg megfelelő karakterére. B: B-t sikeresen illesztette a szöveg megfelelő betűjére; \cancel{B} : sikertelenül illesztette.)

$i =$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
$T[i] =$	A	B	A	B	A	B	A	B	B	A	B	A	B	A	B	B	A	B
	\cancel{B}																	
		<u>B</u>	<u>A</u>	<u>B</u>	<u>A</u>	<u>B</u>	\cancel{B}											
$s=3$				B	A	B	<u>A</u>	<u>B</u>	<u>B</u>	<u>A</u>	<u>B</u>							
									B	A	B	<u>A</u>	<u>B</u>	\cancel{B}				
$s=10$											B	A	B	<u>A</u>	<u>B</u>	<u>B</u>	<u>A</u>	<u>B</u>
																B	A	B

$$S = \{3; 10\}$$

1.2. Jelölések. • Ha x és y két sztring, akkor $x + y$ a konkatenáltjuk.

- Ha x és y két sztring, akkor $x \sqsubseteq y$ (x az y prefixe) azt jelenti, hogy $\exists z$ sztring, amire $x + z = y$.
- Ha x és y két sztring, akkor $x \sqsubset y$ (x az y csonka prefixe) azt jelenti, hogy $x \sqsubseteq y \wedge x \neq y$.
- Ha x és y két sztring, akkor $x \sqsupseteq y$ (x az y szuffixe) azt jelenti, hogy $\exists z$ sztring, amire $z + x = y$.
- Ha x és y két sztring, akkor $x \sqsupset y$ (x az y csonka szuffixe) azt jelenti, hogy $x \sqsupseteq y \wedge x \neq y$.
- $P_j = P[1..j]$ (csak ebben az alfejezetben) P_j a P sztring j hosszúságú prefixe, azaz kezdőszelete. P_0 az üres prefixe. Hasonlóan $T_i = T[1..i]$.
[Eszerint minden nemüres sztringnek csonka szuffixe az üres sztring, azaz $P_0 \sqsupset P_j \quad (j \in 1..m)$.]
- $x \sqsubset y$ (x az y csonka prefix-szuffixe) azt jelenti, hogy $x \sqsubset y \wedge x \sqsupset y$.
- $\max_i H$ a H halmaz i -edik legnagyobb eleme $(i \in 1..|H|)$.
[Ezért $\max_1 H = \max H$, és
ha H véges halmaz, akkor $\max_{|H|} H = \min H$.]
- $H(j) = \{ h \in 0..j-1 \mid P_h \sqsupset P_j \} \quad (j \in 1..m)$
[$0 \in H(j)$, $\max_1 H(j) = \max H(j)$, $\max_{|H(j)|} H(j) = \min H(j) = 0$.]
[Másképpen leírva: $H(j) = \{ |x| : x \sqsubset P_j \} \quad (j \in 1..m)$]
- $next(j) = \max H(j) \quad (j \in 1..m)$

1.3. Tulajdonság. $x \sqsubset y \wedge y \sqsubset z \Rightarrow x \sqsubset z$

1.4. Tulajdonság. $x \sqsubset z \wedge y \sqsubset z \wedge |x| < |y| \Rightarrow x \sqsubset y$

1.5. Tulajdonság. $0 \leq h < j \leq m$ és $P_j \sqsupseteq T_i$ esetén
 $P_h \sqsubset T_i \iff P_h \sqsubset P_j$.

1.6. Tulajdonság. $P_h \supseteq T_i \wedge P[h+1] = T[i+1] \iff P_{h+1} \supseteq T_{i+1}$.

1.7. Tulajdonság. $next(j) \in 0..(j-1) \quad (j \in 1..m)$

1.8. Tulajdonság. $next(j+1) \leq next(j) + 1 \quad (j \in 1..m-1)$
(A $next(j)$ függvény legfeljebb egyesével növekszik.)

1.9. Példa.

$P[j] =$	B	A	B	A	B	B	A	B
$j =$	1	2	3	4	5	6	7	8
$next(j) =$	0	0	1	2	3	1	2	3

Bizonyítás. (Az 1.8. Tulajdonságé)

- $next(j+1) = 0$ esetén $next(j+1) = 0 \leq 1 \leq next(j) + 1$.
- $next(j+1) > 0$ esetén pedig, $next(j+1) = \max H(j+1)$ miatt $next(j+1) \in H(j+1)$, ebből $H(j+1) = \{ h \in 0..j \mid P_h \supset P_{j+1} \}$ szerint $P_{(next(j+1)-1)+1} = P_{next(j+1)} \supset P_{j+1}$, amiből az 1.6. Tulajdonság alapján $P_{next(j+1)-1} \supset P_j$, ahonnan a $next(j) = \max\{ h \in 0..j-1 \mid P_h \supset P_j \}$ meghatározást felhasználva adódik $next(j+1) - 1 \leq next(j)$, végül pedig $next(j+1) \leq next(j) + 1$.

□

1.10. Lemma. $next(\max_l H(j)) \in H(j) \quad (j \in 1..m, l \in 1..|H(j)|-1)$

Bizonyítás.

$P_{next(\max_l H(j))} \supset P_{\max_l H(j)}$, ui. $P_{next(i)} \supset P_i \quad (i \in 1..m)$
 $P_{\max_l H(j)} \supset P_j$. Ezekből a \supset reláció tranzitivitása miatt (1.3)
 $P_{next(\max_l H(j))} \supset P_j$ adódik. Így $next(\max_l H(j)) \in H(j)$. □

1.11. Tulajdonság.

$\max_{l+1} H(j) = next(\max_l H(j)) \quad (j \in 1..m, l \in 1..|H(j)|-1)$

Bizonyítás.

- Először azt látjuk be, hogy $\max_{l+1} H(j) \leq next(\max_l H(j))$.
Nyilván $P_{\max_{l+1} H(j)} \supset P_j \wedge P_{\max_l H(j)} \supset P_j \wedge \max_{l+1} H(j) < \max_l H(j)$.
Ebből 1.4, azaz $x \supset z \wedge y \supset z \wedge |x| < |y| \Rightarrow x \supset y$ szerint
 $P_{\max_{l+1} H(j)} \supset P_{\max_l H(j)}$. Másrészt $P_{\max_{l+1} H(j)} \sqsubset P_{\max_l H(j)}$, azaz
 $P_{\max_{l+1} H(j)} \sqsubset P_{\max_l H(j)}$. Továbbá $P_{next(\max_l H(j))} \sqsubset P_{\max_l H(j)}$, és a $next$ függvény definíciója szerint az ilyen tulajdonságú sztringek között a leghosszabb, tehát $\max_{l+1} H(j) \leq next(\max_l H(j))$.

- Igazolnunk kell még, hogy $next(\max_l H(j)) \leq \max_{l+1} H(j)$. Ehhez $H(j, l) := \{i \in H(j) \mid i < \max_l H(j)\}$. Eszerint $\max_{l+1} H(j) = \max H(j, l)$. Az 1.10. Lemma szerint pedig $next(\max_l H(j)) \in H(j)$. A $next$ függvény definíciójából viszont $next(\max_l H(j)) < \max_l H(j)$. Így $next(\max_l H(j)) \in H(j, l)$, és előbb láttuk, hogy $\max_{l+1} H(j) = \max H(j, l)$. Ezekből már közvetlenül adódik $next(\max_l H(j)) \leq \max_{l+1} H(j)$.

□

1.12. Definíció.

$$\begin{aligned}
 next^1(j) &= next(j) \quad (j \in 1..m) \\
 next^{k+1}(j) &= next(next^k(j)) \quad (next^k(j) \in 1..m) \\
 Szemléletesen: next^k(j) &= \underbrace{next(\dots next(j) \dots)}_k
 \end{aligned}$$

1.13. Tétel. $next^k(j) = \max_k H(j) \quad (k \in 1..|H(j)|)$

Bizonyítás. (Teljes indukcióval)

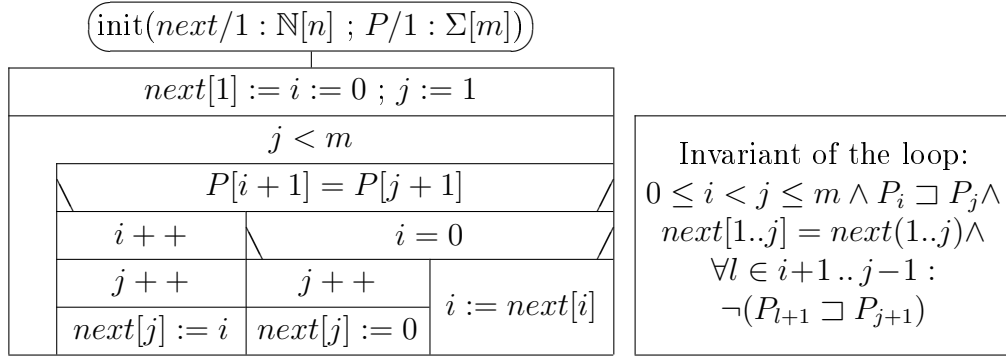
- $k = 1 \Rightarrow next^k(j) = next(j) = \max H(j) = \max_k H(j)$.
- Tegyük fel, hogy $k = l$ -re igaz az állítás ($l \in 1..|H(j)| - 1$).
 $k = l + 1 \Rightarrow next^k(j) = next(next^l(j)) = next(\max_l H(j)) = \max_{l+1} H(j) = \max_k H(j)$.

□

A fenti tétel és tulajdonságok segítségével $\Theta(m)$ műveletigénnyel adunk megoldást a $next/1 : \mathbb{N}[n]$ tömb inicializálására, azzal az utófeltétellel, hogy $\forall j \in 1..m : next[j] = next(j)$, vagy rövidebben: $next[1..m] = next(1..m)$. Az alábbi $init(next, P)$ eljárás tehát a $next/1 : \mathbb{N}[n]$ tömböt a $P/1 : \Sigma[m]$ tömb alapján tölti fel.

A lineáris, pontosabban $\Theta(m)$ műveletigény igazolásához elég belátni, hogy az $init(next, P)$ eljárás ciklusa legalább $m-1$ és legfeljebb $2m-2$ iterációt hajt végre. A ciklus alábbi invariánsa szerint $0 \leq i < j \leq m$.

- Mivel az első iteráció előtt $j = 1$, mindegyik iteráció legfeljebb eggyel növeli j -t, és a ciklusfeltétel $j < m$, legalább $m-1$ iteráció szükséges ahhoz, hogy $j = m$ legyen, és az eljárás befejeződjék.
- $t(i, j) := 2j - i$. Világos, hogy $0 \leq i < j \leq m$ miatt $t(i, j) \in 2..2m$. Az első iteráció előtt $t(i, j) = 2$, mindegyik iterációval (mind a három programágon) szigorúan monoton nő, és végig $\leq 2m$, így legfeljebb $2m-2$ iteráció után megáll a ciklus.



Az $\text{init}(next, P)$ algoritmus szemléltetése az $ABABBABA$ mintán:
 (A három programág mindegyikének az elején kezdünk új sort.)

i	j	$next[j]$	¹ A	² B	³ A	⁴ B	⁵ B	⁶ A	⁷ B	⁸ A
0	1	0		A						
0	2	0			<u>A</u>					
1	3	1			<u>A</u>	<u>B</u>				
2	4	2			<u>A</u>	<u>B</u>	A			
0	4	2					A			
0	5	0						<u>A</u>		
1	6	1						<u>A</u>	<u>B</u>	
2	7	2						<u>A</u>	<u>B</u>	<u>A</u>
3	8	3								

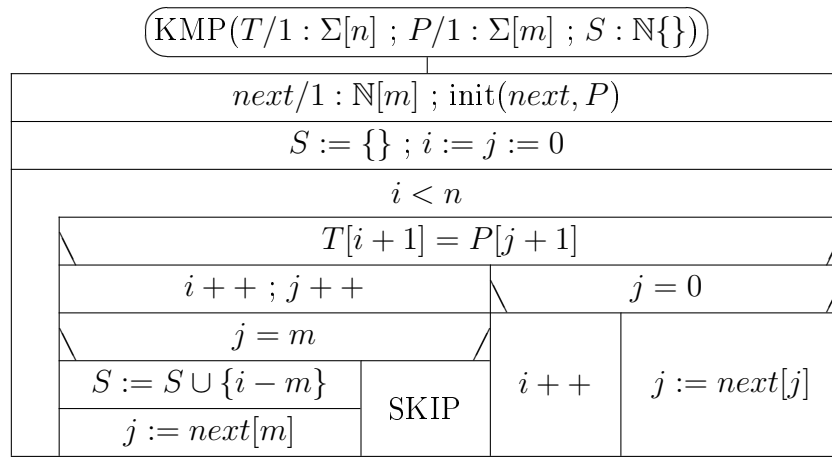
A végeredmény:

$P[j] =$	A	B	A	B	B	A	B	A
$j =$	1	2	3	4	5	6	7	8
$next[j] =$	0	0	1	2	0	1	2	3

Az 1.13. tétel és a fentebbi tulajdonságok segítségével $\Theta(n)$ műveletigénnyel oldjuk meg a mintaillesztési feladatot. Először inicializáljuk a $next/1 : \mathbb{N}[n]$ tömböt, majd ennek segítségével határozzuk meg a minta megfelelő eltolásait.

A lineáris, pontosabban $\Theta(n)$ műveletigény igazolásához elég belátni, hogy az alábbi $KMP(T, P, S)$ eljárás fő ciklusának futási ideje $\Theta(n)$, ui. $m \in 1..n$ miatt ezen az $\text{init}(next, P)$ eljárás és egyéb inicializálások $\Theta(m)$ műveletigénye aszimptotikus nagyságrendben már nem módosít. A fő ciklus $\Theta(n)$ műveletigényének igazolásához pedig elegendő azt belátni, hogy az legalább n és legfeljebb $2n$ iterációt hajt végre. A ciklus alábbi invariánsa szerint $i \in 0..n \wedge j \in 0..(m-1) \wedge j \leq i$.

- Mivel az első iteráció előtt $i = 0$, mindegyik iteráció legfeljebb eggyel növeli i -t, és a ciklusfeltétel $i < n$, legalább n iteráció szükséges ahhoz, hogy $i = n$ legyen, és az eljárás befejeződjék.
- Most $t(i, j) := 2i - j$. Az $i \in 0..n \wedge j \in 0..(m-1) \wedge j \leq i$ invariáns tulajdonság miatt $t(i, j) \in 0..2n$. Az első iteráció előtt $t(i, j) = 0$, mindegyik iterációval (mind a négy programágon) szigorúan monoton nő, és végig $\leq 2n$, így legfeljebb $2n$ iteráció után megáll a ciklus.



Az eljárás utófeltétele: $S = \{s \in 0..(n-m) \mid T[s+1..s+m] = P_m\}$.

Ciklusának invariánsa:

$$\begin{aligned}
& i \in 0..n \wedge j \in 0..(m-1) \wedge j \leq i \wedge \\
& S = \{s \in 0..(i-m) \mid T[s+1..s+m] = P_m\} \wedge \\
& P_j \supseteq T_i \wedge (\forall l \in (j+1)..(m-1) : \neg(P_{l+1} \supseteq T_{i+1})).
\end{aligned}$$

Mint látjuk, a $T[1..n]$ szövegen sohasem kell visszalépni. Ezért ez a KMP algoritmus kényelmesen, hatékonyan implementálható abban az esetben is, ha a szöveg (amiben keressük a mintát) egy szekvenciális fájlban van. Mivel a Brute-force és a Quicksearch algoritmusoknál a szövegen esetleg $m-2$ karakternyit is vissza kell lépni, ezeknél – feltéve, hogy a szöveg egy szekvenciális input fájlban van – annak az utoljára beolvasott $m-1$ karakterét folyamatosan nyilván kell tartani egy átmeneti tárolóban.

Példa:

A $P[1..8] = ABABBABA$ mintát keressük

a $T[1..17] = ABABABBABABBABABA$ szövegben.

A mintához tartozó $next/1 : \mathbb{N}[8]$ tömböt fentebb már kiszámoltuk:

$P[j] =$	A	B	A	B	B	A	B	A
$j =$	1	2	3	4	5	6	7	8
$next[j] =$	0	0	1	2	0	1	2	3

A keresés:

$i =$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
$T[i] =$	A	B	A	B	A	B	B	A	B	A	B	B	A	B	A	B	A
	<u>A</u>	<u>B</u>	<u>A</u>	<u>B</u>	B												
$s=2$			A	B	<u>A</u>	<u>B</u>	<u>B</u>	<u>A</u>	<u>B</u>	<u>A</u>							
$s=7$								A	B	A	<u>B</u>	<u>B</u>	<u>A</u>	<u>B</u>	<u>A</u>		
													A	B	A	<u>B</u>	B
															A	B	<u>A</u>

$$S = \{2; 7\}$$

2. Információtömörítés ([5] 5; [4])

2.1. Naiv módszer

A tömörítendő szöveget karakterenként, fix hosszúságú bitsorozatokkal kódoljuk.

$\Sigma = \langle \sigma_1, \sigma_2, \dots, \sigma_d \rangle$ az ábécé.

Egy-egy karakter $\lceil \lg d \rceil$ bittel kódolható, ui. $\lceil \lg d \rceil$ biten $2^{\lceil \lg d \rceil}$ különböző bináris kód ábrázolható, és $2^{\lceil \lg d \rceil} \geq d > 2^{\lceil \lg d \rceil - 1}$, azaz $\lceil \lg d \rceil$ biten ábrázolható d -féle különböző kód, de eggyel kevesebb biten már nem.

$In : \Sigma^*$ a tömörítendő szöveg. $n = |In|$ jelöléssel $n * \lceil \lg d \rceil$ bittel kódolható.

Pl. az ABRAKADABRA szövegre $d = 5$ és $n = 11$, ahonnét a tömörített kód hossza $11 * \lceil \lg 5 \rceil = 11 * 3 = 33$ bit. (A 3-bites kódok közül tetszőleges 5 kiosztható az 5 betűnek.) A tömörített fájl a kódtáblázatot is tartalmazza.

A fenti ABRAKADABRA szöveg kódtáblázata lehet pl. a következő:

karakter	kód
A	000
B	001
D	010
K	011
R	100

A fenti kódtáblázattal a tömörített kód a következő lesz:

000001100000011000010000001100000.

Ez a tömörített fájlba foglalt kódtáblázat alapján könnyedén 3 bites szakaszokra bontható és kitömöríthető. A kódtáblázat mérete miatt a gyakorlatban csak hosszabb szövegeket érdemes így tömöríteni.

2.2. Huffman-kód

A tömörítendő szöveget karakterenként, változó hosszúságú bitsorozatokkal kódoljuk. A gyakrabban előforduló karakterek kódja rövidebb, a ritkábban előfordulóké hosszabb.

Prefix-mentes kód: Egyetlen karakter kódja sem prefixe semelyik másik karakter kódjának sem.

A karakterenként kódoló tömörítések között a Huffman-kód hossza minimális. Ugyanahhoz a szöveghez többféle kódfa és hozzátartozó kódtáblázat építhető, de mindegyik segítségével az input szövegnek ugyanolyan hosszú tömörített kódját kapjuk. Betömörítés a kódtáblával, kitömörítés a kódfával. Ezért a tömörített fájl a kódfát is tartalmazza.

A tömörítendő fájl, illetve szöveget kétszer olvassa végig.

- Először meghatározza a szövegben előforduló karakterek halmazát és az egyes karakterek gyakoriságát, majd ennek alapján kódfát, abból pedig kódtáblázatot épít.
- Másodszor a kódtábla alapján kiírja az output fájlba sorban a karakterek bináris kódját.

A **kódfa** szigorúan bináris fa. Mindegyik karakterhez tartozik egy-egy levele, amit a karakteren kívül annak gyakorisága, azaz előfordulásainak száma is címkéz. A belső csúcsokat a csúcshoz tartozó részfa leveleit címkéző karakterek gyakoriságainak összegével címkézzük. (Így a kódfa gyökerét a tömörítendő szöveg hossza címkézi.)

A **kódfát úgyépítjük** fel, hogy először egycsúcsú fák egy minimum-prioritásos sorát határozzuk meg, amelyben mindegyik karakter pontosan egy csúcsot címkéz. A csúcsot a karakteren kívül annak gyakorisága is címkézi. A minimum-prioritásos sort a benne tárolt fák gyökerét címkéző gyakoriságértékek szerint építjük fel. Ezután a következőt csináljuk ciklusban, amíg a kupac még legalább kettő fából áll.

Kiveszünk a kupacból egy olyan fát, amelyeknek gyökerét a legkisebb gyakoriság címkézi. Ezután a maradék kupacra ezt még egyszer megismételjük. Összeadjuk a két gyakoriságot. Az összeggel címkézzük egy új csúcsot, amelynek bal és jobb részfája az előbb kiválasztott két fa lesz. A bal ágat a 0, a jobb ágat az 1 címkézi. Az így képzett új fát visszatesszük a minimum-prioritásos sorba.

A fenti ciklus után a minimum-prioritásos sorban maradó egyetlen bináris fa a Huffman-féle kódfa.

A kódfából ezután kódtáblázatot készítünk. Mindegyik karakterekhez tartozó kódot úgy kapjuk meg, hogy a kódfa gyökerétől elindulva és a karakterhez tartozó levél felé haladva a kódfa éleit címkéző biteket összeolvassuk. (Ezt hatékonyan kivitelezhetjük pl. a kódfa preorder bejárásával, az aktuális csúcshoz vezető bitsorozat folyamatos nyilvántartásával, és levélhez érve, a kódtáblázatba írásával.)

Befejezésül újra végigolvassuk a tömörítendő szöveget, és a kódtáblázat segítségével sorban mindegyik karakter bináris kódját a (kezdetben üres) tömörített bitsorozat végéhez fűzzük. A tömörített fájl a kódfát is tartalmazza,

így a gyakorlatban Huffman-kódolással is csak hosszabb szövegeket érdemes tömöríteni.

A **kitömörítést** is karakterenként végezzük. Mindegyik karakter kinyeréséhez a kódfa gyökerétől indulunk, majd a tömörített kód sorban olvasott bitjeinek hatására 0 esetén balra, 1 esetén jobbra lépünk lefelé a fában, míg nem levélcúcshoz érünk. Ekkor kiírjuk a levelet címkéző karaktert, majd a Huffman-kódban a következő bittől és újra a kódfa gyökerétől folytatjuk, amíg a tömörített kódon végig nem érünk.

2.2.1. Huffman-kódolás szemléltetése

Pl. az *ABRAKADABRA* szöveget egyszer végigolvasva meghatározhatjuk milyen karakterek fordulnak elő a szövegben, és milyen gyakorisággal. Úgy képzelhetjük, hogy az alábbi táblázat az új betűkkel folyamatosan bővül, ahogy haladunk előre a szövegben.

szöveg:	A	B	R	A	K	A	D	A	B	R	A
<i>A</i>	1			2		3		4			5
<i>B</i>	-	1							2		
<i>D</i>	-	-	-	-	-	-	1				
<i>K</i>	-	-	-	-	1						
<i>R</i>	-	-	1							2	

A fenti számolást (betű/gyakoriság) alakban összegezve:

$$\langle (D/1), (K/1), [B/2], \{R/2\}, (A/5) \rangle$$

A fenti öt kifejezést öt egycsúcsú bináris fának tekinthetjük. (A jobb olvashatóság kedvéért többféle zárójelpárt alkalmaztunk.) Mindegyik csúcs egyben levél és gyökér. A levelekhez tartozó két címkét *karakter/gyakoriság* alakban írtuk le. A tömörítés algoritmus szerint ezeket egy minimum-prioritásos sorba tesszük. A könnyebb érthetőség kedvéért ezt a minimum-prioritásos sort a szokásos minimum-kupacos reprezentáció helyett most a fák gyökerében lévő gyakoriság-értékek (röviden *fa-gyakoriság-értékek*) szerint rendezett fa-sorozattal szemléltetjük. (Azonos gyakoriságok esetén a betűk alfabetikus sorrendje szerint rendezünk. Ez ugyan önkényes, de az algoritmus bemutatása szempontjából hasznos egyértelműsítés. A fák ágait is hasonlóképpen rendezzük sorba.)

Ezután kivesszük a két legkisebb gyakoriság-értékű fát, egy új gyökércsúcs alá tesszük őket bal- és jobboldali részfának, a új gyökércsúcsot pedig a két

fa-gyakoriság-érték összegével címkézzük. Végül visszatesszük az új fát a minimum-prioritásos sorba.

$$\langle [B/2], [(D/1)2(K/1)], \{R/2\}, (A/5) \rangle$$

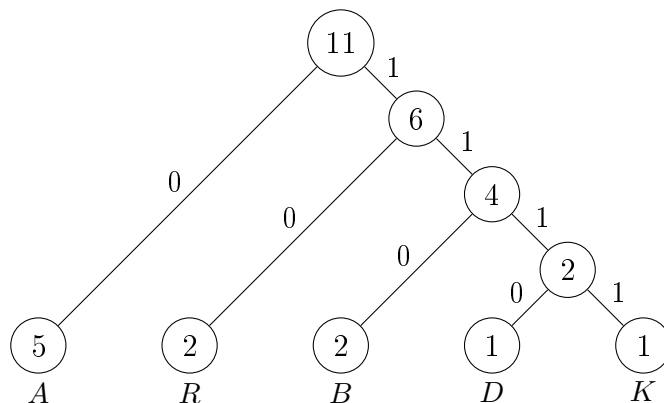
A fenti eljárást addig ismételjük, amíg már csak egy fánk marad. Ezt végül kivesszük a minimum-prioritásos sorból: ez a Huffman-féle kódfa.

$$\langle \{R/2\}, \{[B/2]4[(D/1)2(K/1)]\}, (A/5) \rangle$$

$$\langle (A/5), (\{R/2\}6\{[B/2]4[(D/1)2(K/1)]\}) \rangle$$

$$[(A/5)11(\{R/2\}6\{[B/2]4[(D/1)2(K/1)]\})]$$

A fent kapott kódfát az 1. ábrán is láthatjuk.



1. ábra. Az *ABRAKADABRA* szövegnek az alfabetikus konvencióval adódó Huffman-féle kódfája

Tekintsünk az 1. ábrán látható kódfában egy tetszőleges egyszerű, azaz körmentes utat, amely a fa gyökerétől lefelé valamelyik leveléig halad! Az út éleit címkéző biteket összeolvasva adódik a levelet címkéző karakter Huffman-kódja. Így a karakterekre a következő kódtáblázatot kapjuk.

karakter	kód
<i>A</i>	0
<i>B</i>	110
<i>D</i>	1110
<i>K</i>	1111
<i>R</i>	10

A fentiek alapján az ABRAKADABRA szöveg Huffman kódja 23 bit, ami lényegesen rövidebb, mint a fenti naiv tömörítés esetén. A kódtáblázat bináris kódjait az ABRAKADABRA szöveg karakterei szerint sorban egymás után fűzve kapjuk a szöveg Huffman-kódját.

01101001111011100110100

A **kitömörítéshez** az előbbi Huffman-kód és a kódfa alapján a kezdő nulla rögtön az „A” címkéjű levélhez visz. Ezután sorban olvasva a maradékból a biteket, a 110 a B-hez visz, majd a 10 az R-hez, a 0 az A-hoz, a 1111 a K-hoz, a 0 az A-hoz, az 1110 a D-hez, a 0 az A-hoz, a 110 a B-hez, a 10 az R-hez, és végül a 0 az A-hoz. Így visszakaptuk az eredeti, tömörítetlen szöveget.

2.1. Feladat. *Próbáljuk ki, hogy ha a Huffman-kódolásban lévő indeterminizmusokat a fenti alfabetikus sorrendtől eltérően oldjuk fel, ugyanarra a tömörítendő szövegre mégis mindig ugyanolyan hosszú Huffman-kódot kapunk! (Ha például a minimum-prioritásos sorból azonos fa-gyakoriság-értékek esetén az alacsonyabb fát vesszük ki előbb – ezt az ad-hoc szabályt az alfabetikus konvencionál erősebbnek véve –, akkor a fenti példában a kódfát felépítő ciklus második iterációjában a $[B/2]$ és az $\{R/2\}$ fát fogjuk összevonni.)*

2.3. Lempel–Ziv–Welch (LZW) módszer

Az input szöveget ismétlődő mintákra (sztringekre) bontja. Mindegyik mintát ugyanolyan hosszú bináris kóddal helyettesíti. Ezek a minták kódjai. A tömörített fájl a kódtáblázatot nem tartalmazza. Részletes magyarázat olvasható Ivanyos Gábor, Rónyai Lajos és Szabó Réka: *Algoritmusok* c. könyvében [5]. (Online elérhetősége az irodalomjegyzékünkben.)

Jelölések az absztrakt struktogramokhoz:

- Ha a kódok b bitesek, akkor $MAXCODE = 2^b - 1$ globális konstans a kódként használható legnagyobb számérték. Ha pl. $b = 12$, akkor $MAXCODE = 2^{12} - 1 = 4095$.
- A $\Sigma = \langle \sigma_1, \sigma_2, \dots, \sigma_d \rangle$ sorozat tartalmazza az ábécé karaktereit.
- A tömörítésnél „In” a tömörítendő szöveg. „Out” a tömörítés eredménye: kódok sorozata. A kitömörítésnél fordítva.
- D a szótár, ami $(string, code)$ rendezett párok, azaz $Item$ -ek halmaza. A szótárat a tömörített kód nem tartalmazza. Ehelyett a kitömörítés rekonstruálja az ábécé és a tömörített kód alapján.

$Item$
$+string : \Sigma^{\langle \rangle}$
$+code : \mathbb{N}$
$+Item(s : \Sigma^{\langle \rangle} ; k : \mathbb{N}) \{ string := s ; code := k \}$

$\text{LZWcompress}(In : \Sigma\langle \rangle ; Out : \mathbb{N}\langle \rangle)$		
$D : Item\{\}$ // D is the dictionary, initially empty		
$i := 1$ to $ \Sigma $		
	$x : Item(\langle \Sigma_i \rangle, i) ; D := D \cup \{x\}$	
$code := \Sigma + 1 ; Out := \langle \rangle ; s : \Sigma\langle In_1 \rangle$		
$i := 2$ to $ In $		
	$c : \Sigma := In_i$	
	dictionaryContainsString($D, s + c$)	
$s := s + c$	$Out := Out + code(D, s)$	
	$code \leq MAXCODE$	
	$x : Item(s + c, code++) ; D := D \cup \{x\}$	SKIP
	$s := \langle c \rangle$	
$Out := Out + code(D, s)$		

(LZWdecompress($In : \mathbb{N}\langle \rangle$; $Out : \Sigma\langle \rangle$))	
$D : Item\{\}$ // D is the dictionary, initially empty	
$i := 1$ to $ \Sigma $	
$x : Item(\langle \Sigma_i \rangle, i)$; $D := D \cup \{x\}$	
$code := \Sigma + 1$ // code is the first unused code	
$Out := s := string(D, In_1)$	
$i := 2$ to $ In $	
$k := In_i$	
$k < code$ // D contains k	
$t := string(D, k)$	$t := s + s_1$
$Out := Out + t$	$Out := Out + t$
$x : Item(s + t_1, code)$ $D := D \cup \{x\}$	$x : Item(t, k)$ // k=code $D := D \cup \{x\}$
$s := t$; $code++$	