

# Záróvizsga tételsor

## 14. Haladó algoritmusok

Dobreff András

### Haladó algoritmusok

Gráfalgoritmusok: gráfok ábrázolás, szélességi bejárás, minimális költségű utak keresése, minimális költségű feszítőfa keresése, mélységi bejárás, DAG topologikus rendezése. Adattömörítések (Huffman- és LZW-algoritmus). Mintaillesztés módszerei.

## 1 Gráfalgoritmusok

### 1.1 Gráf ábrázolás

#### Láncolt listás ábrázolás

A gráf csúcsait helyezzük el egy tömbben (vagy láncolt listában). Minden elemhez rendeljünk hozzá egy láncolt listát, melyben az adott csúcs szomszédjait (az esetleges élsúlyokkal) soroljuk fel.

#### Mátrixos ábrázolás

Legyen a csúcsok elemszáma  $n$ . Ekkor egy  $A^{n \times n}$  mátrixban jelöljük, hogy mely csúcsok vannak összekötve. Ekkor mind a sorokban, mind az oszlopokban a csúcsok szerepelnek, és az  $a_{ij}$  cellában a  $i$  csúcsból  $j$  csúcsba vezető él súlya szerepel, ha nincs él a két csúcs között, akkor  $-\infty$  (súlyozatlan esetben 1 és 0)

Amennyiben a gráf irányítatlan nyilván  $a_{ij} = a_{ji}$

### 1.2 Szélességi bejárás

$G$  gráf (irányított/irányítatlan)  $s$  startcsúcsából a távolság sorrendjében érjük el a csúcsokat. A legrövidebb utak feszítőfáját adja meg, így csak a távolság számít, a súly nem.

A nyilvántartott csúcsokat egy sor adatszerkezetben tároljuk, az aktuális csúcs gyerekeit a sor-ba tesszük. A következő csúcs pedig a sor legelső eleme lesz.

A csúcsok távolságát egy  $d$ , szüleit egy  $\pi$  tömbbe írjuk, és  $\infty$  illetve 0 értékekkel inicializáljuk.

Az algoritmus:

1. Az  $s$  startcsúcsot betesszük a sorba
2. A következő lépéseket addig ismétljük, míg a sor üres nem lesz
3. Kivesszük a sor legelső ( $u$ ) elemét
4. Azokat a gyerekcsúcsokat, melyeknek a távolsága nem  $\infty$  figyelmen kívül hagyjuk (ezeken már jártunk)
5. A többi gyerekre ( $v$ ): beállítjuk a szülőjét ( $\pi[v] = u$ ), és a távolságát ( $d[v] = d[u] + 1$ ). Majd berakjuk a sorba.

### 1.3 Minimális költségű utak keresése

#### Dijkstra algoritmus

Egy  $G$  irányított, pozitív élsúlyokkal rendelkező gráfban keres  $s$  startcsúcsból minimális költségű utakat minden csúcsához.

Az algoritmus a szélességi bejárás módosított változata. Mivel itt egy hosszabb útnak lehet kisebb a költsége, mint egy rövidebbnek, egy már megtalált csúcsot nem szabad figyelmen kívül hagyni. Ezért minden csúcs rendelkezik három állapottal (nem elért, elért, kész). A  $d$  és  $\pi$  tömböket a szélességi bejáráshoz hasonlóan kezeljük.

A még nem kész csúcsokat egy prioritásos sorba helyezzük, vagy minden esetben minimumkeresést alkalmazunk.

Az algoritmus:

1. Az  $s$  startcsúcs súlyát 0-ra állítjuk eltároljuk
2. A következő lépéseket addig ismétljük, míg a konténerünk üres nem lesz
3. Kivesszük a sor legjobb ( $u$ ) elemét, és "kész"-re állítjuk
4. Ha egy gyerekcsúcs ( $v$ ) nem kész, és a jelenleg hozzávezető út súlya kisebb, mint az eddigi, akkor: a szülőjét  $u$ -ra állítjuk ( $\pi[v] = u$ ), és a súlyát frissítjük ( $d[v] = d[u] + d(u, v)$ ).
5. A többi csúcsot kihagyjuk.

### Bellman-Ford algoritmus

Egy  $G$  élsúlyozott (akár negatív) irányított gráf  $s$  startcsúcsából keres minden élhez minimális költségű utakat, illetve felismeri, ha negatív költségű kör van a gráfban. A  $d$  és  $\pi$  tömböket az előzőekhez hasonlóan kezeljük.

Az algoritmus:

1. A startcsúcs súlyát állítsuk be 0-ra.
2.  $n-1$  iterációban menjünk végig az összes csúcson, és minden csúcsot ( $u$ ) vessünk össze minden csúccsal ( $v$ ). Ha olcsóbb utat találtunk akkor  $v$ -be felülírjuk a súlyát ( $d[v] = d[u] + d(u, v)$ ), és a szülőjét ( $\pi[v] = u$ ).
3. Ha az  $n$ -edik iterációban is történt módosítás, negatív kör van a gráfban

## 1.4 Minimális költség feszítőfa keresése

A Prim algoritmus egy irányítatlan élsúlyozott (akár negatív) gráf  $s$  startcsúcsából keres minimális költségű feszítőfát. A  $d$  és  $\pi$  tömböket az előzőekhez hasonlóan kezeljük. Az algoritmus egy prioritásos sorba helyezi a csúcsokat.

Az algoritmus:

1. A startcsúcs súlyát állítsuk be 0-ra.
2. A csúcsokat behelyezzük a prioritásos sorba.
3. A következő lépéseket addig végezzük, míg a prioritásos sor ki nem ürül.
4. Kivesszünk egy csúcsot ( $u$ ) a sorból.
5. Minden gyerekére ( $v$ ), amely még a sorban és a nyilvántartott  $v$ -be vezető él súlya nagyobb, mint a most megtalált: A  $v$  szülőjét  $u$ -ra változtatjuk, a nyilvántartott súlyt felülírjuk  $d[v] = d(u, v)$ . Majd felülírjuk a  $v$  állapotát a prioritásos sorban.
6. Azokkal a gyerekekkel, melyek nincsenek a sorban, vagy a súlyukon nem tudunk javítani, nem változtatunk.

## 1.5 Mélységi bejárás

$G$  irányított (nem feltétlenül összefüggő) gráf mélységi bejárásával egy mélységi fát (erdőt) kapunk. Az algoritmus a következő:

- Az élsúlyok nem játszanak szerepet
- Nincs startcsúcs, a gráf minden csúcsára elindítjuk az algoritmust. (Természetesen ekkor, ha már olyan csúcsot választunk, amin már voltunk, az algoritmus nem indul el.)
- A csúcsokat mohón választjuk, azaz minden csúcs gyerekei közül az elsőt választva haladunk előre, amíg csak lehet. (Olyan csúcsot találunk, amelynek nincs gyereke, vagy minden gyerekén jártunk már.)
- Ha már nem lehet előre haladni visszalépünk.

- Minden csúcshoz hozzárendelünk két értéket. Az egyik a mélységi sorszám, mely azt jelöli, hogy hanyadiknak értük el. A másik a befejezési szám, mely azt jelzi, hogy hanyadiknak léptünk vissza belőle.

A gráf éleit a mélységi bejárás közben osztályozhatjuk. (Inicializáláskor minden értéket 0-ra állítottunk)

- Faél: A következő csúcs mélységi száma 0
- Visszaél: A következő csúcs mélységi száma nagyobb, mint 0, és befejezési száma 0 (Tehát az aktuális út egy előző csúcsára kanyarodunk vissza)
- Keresztél: A következő csúcs mélységi száma nagyobb, mint 0, és befejezési száma is nagyobb, mint 0, továbbá az aktuális csúcs mélységi száma nagyobb, mint a következő csúcs mélységi száma. (Ekkor egy az aktuális csúcsot megelőző csúcsból induló, már megtalált útba mutató éllel van dolgunk)
- Előreél: A következő csúcs mélységi száma nagyobb, mint 0, és befejezési száma is nagyobb, mint 0, továbbá az aktuális csúcs mélységi száma kisebb, mint a következő csúcs mélységi száma. (Ekkor egy az aktuális csúcsból induló, már megtalált útba mutató éllel van dolgunk)

## 1.6 DAG Topologikus rendezése

### Alapfogalmak

- Topologikus rendezés:  
Egy  $G(V, E)$  gráf topologikus rendezése a csúcsok olyan sorrendje, melyben  $\forall (u \rightarrow v) \in E$  élre  $u$  előbb van a sorrendben, mint  $v$
- DAG - Directed Acyclic Graph:  
Írányított körmentes gráf.  
Legtöbbször munkafolyamatok irányítására illetve függőségek analizálására használják.  
Tulajdonságok:
  - Ha  $G$  gráfra a mélységi bejárás visszaél talál (Azaz kört talált)  $\implies G$  nem DAG
  - Ha  $G$  nem DAG (van benne kör)  $\implies$  Bármely mélységi bejárás talál visszaél
  - Ha  $G$ -nek van topologikus rendezése  $\implies G$  DAG
  - Minden DAG topologikusan rendezhető.

### DAG topologikus rendezése

Egy  $G$  gráf mélységi bejárása során tegyük verembe azokat a csúcsokat, melyekből visszaléptünk. Az algoritmus után a verem tartalmát kiírva megkapjuk a gráf egy topologikus rendezését.

## 2 Adattömörítések

### 2.1 Huffman-algoritmus

A Huffman-algoritmussal való tömörítés lényege, hogy a gyakrabban előforduló elemeket (karaktereket) rövidebb, míg a ritkábban előfordulókat hosszabb kódszavakkal kódoljuk.

Ehhez tisztában kell lennünk az egyes karakterek gyakoriságával (vagy relatív gyakoriságával). Ezek alapján egy ún. Huffman-fát építünk, melyben az éleket a kód betűivel címkézzük, a fa levelein a kódolandó betűk helyezkednek el, a gyökérből a levelekig vezető út címkei alapján rajuk össze a kódszavakat.

Az algoritmus (spec. bináris Huffman fára):

1. A kódolandó szimbólumokat gyakoriságaik alapján sorba rendezzük.
2. A következő redukciós lépéseket addig hajtjuk végre, míg egy csoportunk marad.
3. Kiválasztjuk az utolsó két elemet (legritkább), összevonjuk őket egy új csoportba, és ennek a csoportnak a gyakorisága a gyakoriságok összege lesz.
4. A csoportot visszahelyezzük a rendezett sorba (gyakoriság alapján rendezve).

5. A csoportból új csúcsot képezünk, mely csúcs az öt alkotó két elem szülője lesz.

Példa:

Legyen a következő 5 betű, mely a megadott gyakorisággal fordul elő:

A	B	C	D	E
5	4	3	2	1

Ekkor a redukciós lépések a következők:

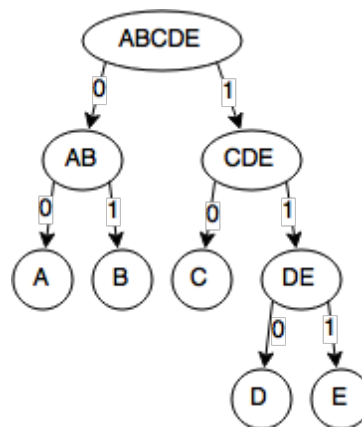
- | A | B | C | D, E |
|---|---|---|------|
| 5 | 4 | 3 | 3    |

- | C, D, E | A | B |
|---------|---|---|
| 6       | 5 | 4 |

- | A, B | C, D, E |
|------|---------|
| 9    | 6       |

- | A, B, C, D, E |
|---------------|
| 15            |

A Huffman-fa a XY. ábrán látható.



ábra 1: Huffman-fa példa

Tehát a kódszavak:

A	B	C	D	E
00	01	10	110	111

## 2.2 LZW-algoritmus

Az LZW (Lempel-Ziv-Welch) tömörítésnek a lényege, hogy egy szótárat bővítünk folyamatosan, és az egyes kódolandó szavakhoz szótárindexeket rendelünk.

### Kódolás

A kódolás algoritmus a következő lépésekből áll:

1. A szótárt inicializáljuk az összes 1 hosszú szóval
2. Kikeressük a szótárból a leghosszabb, jelenlegi inputtal összeillő  $W$  sztringet
3.  $W$  szótárindexét kiadjuk, és  $W$ -t eltávolítjuk az inputról
4. A  $W$  szó és az input következő szimbólumának konkatenációját felvesszük a szótárba
5. A ?? lépéstől ismételjük

## Dekódolás

A dekódolás során is építenünk kell a szótárat. Ezt már azonban csak a dekódolt szöveg(rész) segítségével tudjuk megtenni, mivel egy megkapott kód dekódolt szava és az utána lévő szó első karakteréből áll össze a szótár következő eleme.

Tehát a dekódolás lépései:

1. Kikeressük a kapott kódhoz tartozó szót a szótárból ( $u$ ), az output-ra rakjuk
2. Kikeressük a következő szót ( $v$ ) a szótárból, az első szimbólumát  $u$ -hoz konkatenálva a szótárba rakjuk a következő indexszel.
3. Amennyiben már nincs következő szó, dekódolunk, de nem írunk a szótárba.

Megtörténhet az az eset, hogy mégis kapunk olyan kódszót, mely még nincs benne a szótárban. Ez akkor fordulhat elő, ha a kódolásnál az aktuálisan szótárba írt szó következik.

Példa:

Szöveg: AAA

Szótár: A - 1

Ekkor a kódolásnál vesszük az első karaktert, a szótárbeli indexe 1, ezt kiküldjük az outputra. A következő karakter A, így AA-t beírjuk a szótárba 2-es indexszel. Az első karaktert töröljük az inputról. Addig olvasunk, míg szótárbeli egyezést találunk, így AA-t olvassuk (amit pont az előbb raktunk be), ennek indexe 2, tehát ezt küldjük az outputra. AA-t töröljük az inputról, és ezzel végeztünk is. Az output: 1,2

Dekódoljuk az 1,2 inputot! Jelenleg a szótárban csak A van 1-es indexszel. Vegyük az input első karakterét, az 1-et, ennek szótárbeli megfelelője A. Ezt tegyük az outputra. A következő index a 2, de ilyen bejegyzés még nem szerepel a szótárban.

Ebben az esetben a dekódolásnál, egy trükköt vetünk be. A szótárba írás pillanatában még nem ismert a beírandó szó utolsó karaktere (A példában A-t találtuk, de nem volt 2-es bejegyzés). Ekkor ?-et írunk a szótárba írandó szó utolsó karakterének helyére. (Tehát A? - 2 kerül a szótárba). De mostmár tudni lehet az új bejegyzés első betűjét (A? - 2 az új bejegyzés, ennek első betűje A). Cseréljük le a ?-et erre a betűre. (Tehát AA - 2 lesz a szótárban).

## 3 Mintaillesztés

### 3.1 Knuth-Morris-Pratt algoritmus

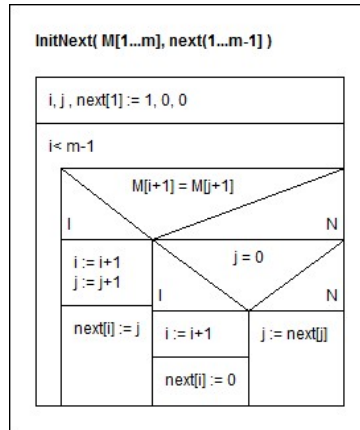
A Knuth-Morris-Pratt eljárásnak a Brute-Force (hasonlítsuk össze, toljunk egyet, stb..) módszerrel szemben az az előnye, hogy egyes esetekben, ha a mintában vannak ismétlődő elemek, akkor egy tolásnál akár több karakternyit is ugorhatunk.

...	A	B	A	B	A	B	A	C	...
	A	B	A	B	A	C			
			A	B	A	B	A	C	

ábra 2: KMP algoritmus több karakter tolás esetén

Az ugrás megállapítását a következőképp tesszük: Az eddig megvizsgált egyező mintarész elején (prefix) és végén (suffix) olyan kartersorozatot keresünk, melyek megegyeznek. Ha találunk ilyen, akkor a mintát annyiival tolhatjuk, hogy az elején lévő része ráilleszkedjen a végén levőre.

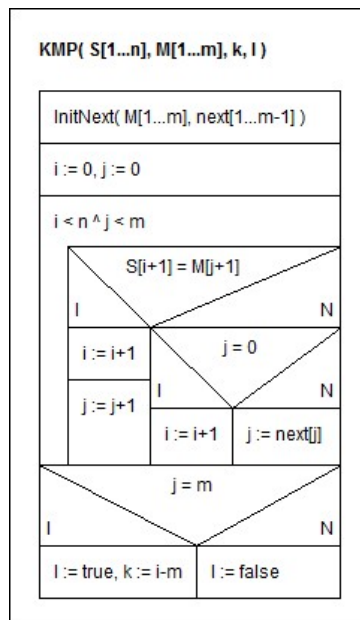
Azt, hogy ez egyes esetekben mekkorát tolhatunk nem kell minden elromlás alkalmával vizsgálni. Ha a mintára önmagával lefuttatjuk az algoritmus egy módosított változatát (?? ábra), kitölthetünk egy tömböt, mely alapján a tolásokat végezni fogjuk.



ábra 3: KMP tolásokat szabályzó tömb kitöltése

Az algoritmus (ld ???. ábra):

- Két indexet  $i$  és  $j$  futtatunk a szövegen illetve a mintán.
- Ha az  $i + 1$ -edik és  $j + 1$ -edik karakterek megegyeznek, akkor léptetjük mind a kettőt.
- Ha nem egyeznek meg, akkor:
  - Ha a minta első elemét vizsgáltuk, akkor egyet tolunk a mintán, magyarul a minta indexe marad az első betűn, és a szövegben lévő indexet növeljük eggyel ( $i = i + 1$ )
  - Ha nem a minta első elemét vizsgáltuk, akkor annyit tolunk, amennyit szabad. Ez azt jelenti, hogy csak a mintán lévő indexet helyezzük egy kisebb helyre ( $j = next[j]$ )
- Addig megyünk, míg vagy a minta, vagy a szöveg végére nem érünk. Ha a minta végére értünk, akkor megtaláltuk a mintát a szövegben, ha a szöveg végére értünk, akkor pedig nem.



ábra 4: KMP algoritmus

### 3.2 Boyer-Moore — Quick search algoritmus

Míg a KMP algoritmus az elomlás helye előtti rész alapján döntött a tolásról, addig a QS a minta utáni karakter alapján. Tehát elomlás esetén:

- Ha a minta utáni karakter benne van a mintában, akkor jobbról az első előfordulására illesztjük. (?? . ábra)

...	A	B	A	A	C	B	C	D	...
	A	A	C	B	C	D			
			A	A	C	B	C	D	

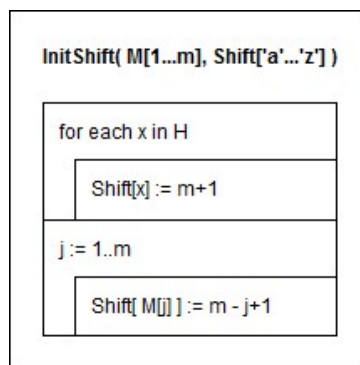
ábra 5: QS - eltolás ha a minta utáni karakter benne van a mintában

- Ha a minta utáni karakter nincs benne a mintában, akkor a mintát ezen karakter után illesztjük. (?? . ábra)

...	A	B	A	A	C	B	X	D	...
	A	A	C	B	C	D			
							A	A	C
							B	C	D

ábra 6: QS - eltolás ha a minta utáni karakter nincs benne a mintában

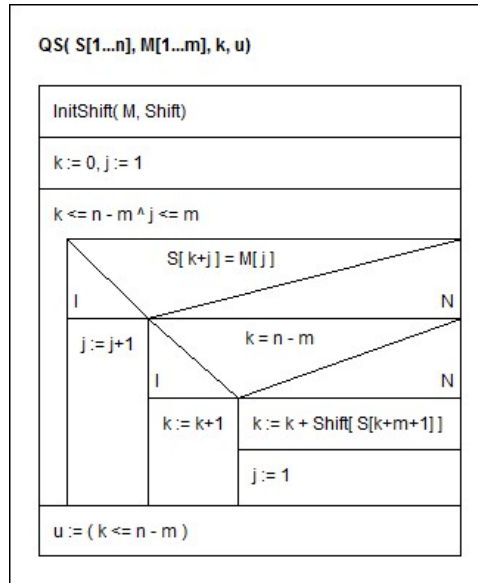
Az eltolás kiszámítását megint elő lehet segíteni egy tömbbel, most azonban, mivel nem a minta az érdekes, és nem tudjuk pontosan mely karakterek szerepelnek a szövegben, így a tömbbe az egész abc-t fel kell vennünk (?? . ábra)



ábra 7: QS - Az eltolást elősegítő tömb ( $Shift['a'...'z']$ ) konstruálása

Az algoritmus (ld. ?? . ábra):

- Két indexet  $k$  és  $j$  futtatunk a szövegen illetve a mintán.
- Ha a szöveg  $k + j$ -edik eleme megegyezik a minta  $j$ -edik karakterével, akkor léptetjük  $j$ -t (mivel a szövegben  $k + j$ -edik elemet nézzük, így elég  $j$ -t növelni).
- Ha nem egyeznek meg, akkor:
  - Ha a minta már a szöveg végén van ( $k = n - m$ ), akkor csak növeljük  $k$ -t eggyel, ami hamissá teszi a ciklus feltételt.
  - Ha még nem vagyunk a szöveg végén  $k$ -t toljuk annyival, amennyivel lehet (ezt az előre beállított  $Shift$  tömb határozza meg). És a  $j$ -t visszaállítjuk 1-re.
- Addig megyünk, míg vagy a minta végére érünk  $j$ -vel, vagy a mintát továbbtoltuk a szöveg végénél. Előbbi esetben egyezést találtunk, míg az utóbbiban nem.



ábra 8: QS

### 3.3 Rabin-Karp algoritmus

A Rabin-Karp algoritmus lényege, hogy minden betűhöz az ábécéből egy számjegyet rendelünk, és a keresést számok összehasonlításával végezzük. Világos, hogy ehhez egy ábécé méretnek megfelelő számrendszerre lesz szükségünk. A szövegből mindig a minta hosszával egyező részeket szelünk ki, és ezeket hasonlítjuk össze.

Példa:

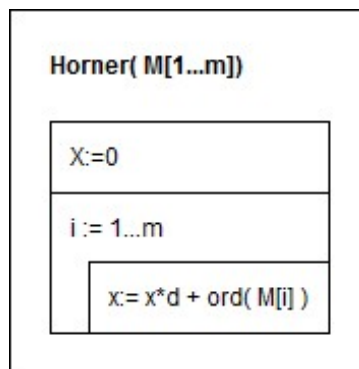
Minta: BBAC  $\rightarrow$  1102

Szöveg: DACABBAC  $\rightarrow$  30201102, amiből a következő számokat állítjuk elő: 3020, 0201, 2011, 0110, 1102

A fent látható szeletek lesznek az  $s_i$ -k.

Az algoritmus működéséhez azonban számos apró ötletet alkalmazunk:

1. A minta számokká alakítását Horner-módszer segítségével végezzük.



ábra 9: RK - Horner-módszer

Az  $\text{ord}()$  függvény az egyes betűknek megfelelő számot adja vissza. A  $d$  a számrendszer alapszáma.

2. A szöveg mintával megegyező hosszú szeleteinek ( $s_i$ ) előállítását:  
 $s_0$ -t a Horner-módszerrel ki tudjuk számolni. Ezek után  $s_{i+1}$  a következőképp számolandó:

$$s_{i+1} = (s_i - \text{ord}(S[i]) \cdot d^{m-1}) \cdot d + \text{ord}(S[i+1])$$



Magyarázat:  $s_i$  elejétől levágjuk az első számjegyet ( $s_i - \text{ord}(S[i]) \cdot d^{m-1}$ ), majd a maradékot eltoljuk egy helyiértékkal (szorzás  $d$ -vel), végül az utolsó helyiértékre beírjuk a következő betűnek megfelelő számjegyet ( $+\text{ord}(S[i+1])$ )

Példa:

Az előző példa szövegével és mintájával ( $d = 10$  elemű ábécé és  $m = 4$  hosszú minta):

$s_0 = 3020$ , ekkor:  $s_{0+1} = s_1 = (3020 - \text{ord}(D) \cdot 10^3) \cdot 10 + \text{ord}(B) = (3020 - 3000) \cdot 10 + 1 = 0201$

3. Felmerülhet a kérdés, hogy az ilyen magas alapszámú számrendszerek nem okoznak-e gondot az ábrázolásnál? A kérdés jogos. Vegyük a következő életszerű példát:

4 bájtton ábrázoljuk a számainkat ( $2^{32}$ ). Az abc legyen 32 elemű ( $d = 32$ ), a minta 8 hosszú ( $m = 8$ ). Ekkor a  $d^{m-1}$  kiszámítása:  $32^7 = (2^5)^7 = 2^{35}$ , ami már nem ábrázolható 4 bájtton.

Ennek kiküszöbölésére vezessünk be egy nagy  $p$  prímét, melyre  $d \cdot p$  még ábrázolható. És a műveleteket számoljuk  $\text{mod } p$ . Ekkor természetesen a kongruencia miatt lesz olyan eset, amikor az algoritmus egyezést mutat, mikor valójában nincs. Ez nem okoz gondot, mivel ilyen esetben karakterenkénti egyezést vizsgálva ezt a problémát kezelni tudjuk. (Fordított eset nem fordul elő tehát nem lesz olyan eset, mikor karakterenkénti egyezés van, de numerikus nincs). [Ha  $p$  kellően nagy, a jelenség nagyon ritkán fordul elő.]

4. A  $\text{mod } p$  számítás egy másik problémát is felvet. Ugyanis a kivonás alkalmával negatív számokat is kaphatunk.

Például: Legyen  $p = 7$ , ekkor, ha  $\text{ord}(S[i]) = 9$ , akkor előző számítás után  $s_i = 2\dots$ , de ebből  $\text{ord}(S[i]) \cdot d^{m-1} = 9 \cdot 10^3 = 9000$ -et vonunk ki negatív számot kapunk.

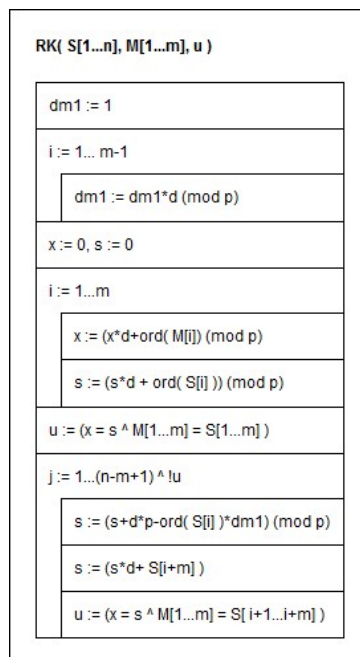
Megoldásként  $s_{i+1}$ -et két lépésben számoljuk:

$$s := (s_i + d \cdot p - \text{ord}(S[i]) \cdot d^{m-1}) \text{ mod } p$$

$$s_{i+1} := (s \cdot d + \text{ord}(S[i+1])) \text{ mod } p$$

A fentiek alapján az algoritmus a következő (ld. ?? ábra)

1. Kiszámoljuk  $d^{m-1}$ -et ( $dm1$ )
2. Egy iterációban meghatározzuk Horner-módszerrel a minta számait ( $x$ ) és  $s_0$ -t
3. Ellenőrizzük, hogy egyeznek-e
4. Addig számolgatjuk  $s_i$  értékét míg a minta nem egyezik  $s_i$ -vel, vagy a minta a szöveg végére nem ért.



ábra 10: RK