MINTAILLESZTÉS **2**

1. Brute-Force algoritmus (BF) *4*
2. Knuth-Morris-Pratt algoritmus (KMP) *7*
3. Quick-Search algoritmus (QS) *13*
4. Rabin-Karp algoritmus (RK) *17*
Felhasznált irodalom *19*

Mintaillesztés

(String keresés)

Feladat: Keressük meg egy szövegben egy szövegminta (szövegrészlet, string) előfordulását vagy előfordulásait.
A **feladat általánosítható:** valamely alaptípus feletti sorozatban keressük egy másik ugyanezen alaptípus feletti sorozat előfordulásait (Pl.: egy DNS láncban keresünk egy szakaszt)
A továbbiakban **egyszerűsítjük a feladatot** a minta **első előfordulásának** a megkeresésére, amelynek segítségével az összes előfordulás megkapható (Keressük meg a minta első előfordulását, majd a hátralévő szövegbe ismét keressük az első előfordulást stb.).

Vezessük be a fejezet egészére az alábbi **jelöléseket**:

- Legyen *H* egy tetszőleges alaptípus feletti véges halmaz, ún. **ABC**.
- Legyen a **szöveg**, amelyben a mintát keressük: $S[1..n] \in H^*$, azaz egy *n* hosszú *H* feletti véges sorozat.
- Legyen a **minta**, amelyet keresünk a szövegben: $M[1..m] \in H^*$, egy *m* hosszú szintén a *H* feletti véges sorozat.

Továbbá tegyük fel, hogy *S*-en és *M*-en megengedett művelet az **indexelés**, azaz hivatkozhatunk a szöveg vagy a minta egy *i*-dik elemére $S[i]$ $(0<i<n+1)$, $M[i]$ $(0<i<m+1)$. A tárgyalt algoritmusok némelyike egy az egyben átírható **szekvenciális fájlokra** is (ahol az indexelés nem megengedett), míg a többi tárgyalt algoritmus csak **buffer** használatával alkalmazható szekvenciális fájlokra.

Definíció:
Legyen $k \in N, k \in [0..n-m]$

- Az *M* $k+1$-dik **pozíción illeszkedik** (előfordul) az *S* szövegre (szövegben)
- Az *M* *k* **eltolással illeszkedik** (előfordul) *S*-re (*S*-ben)
- *k* **érvényes eltolás**

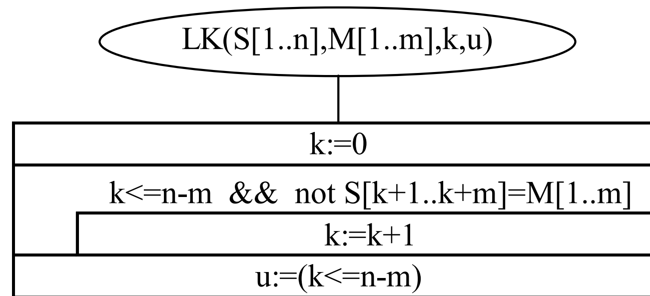
, ha $S[k+1..k+m]=M[1..m]$, azaz $\forall j \in N,\ j \in [1..m] : S[k+j] = M[j]$
Továbbá az *M* $k+1$-dik pozíción való illeszkedése az *M* **első előfordulása** az *S* szövegben, ha $\forall i \in N, i \in [0..k-1] : S[i+1..i+m] \neq M[1..m]$

Pl.: Legyen a szövegünk S="ABBABCAB", és a keresett mintánk M="ABC". A fenti definíció szerint az *M* minta a 4-dik pozíción, *k*=3 eltolással illeszkedik az *S* szövegre.

A	B	B	A	B	C	A	B
			A	B	C		

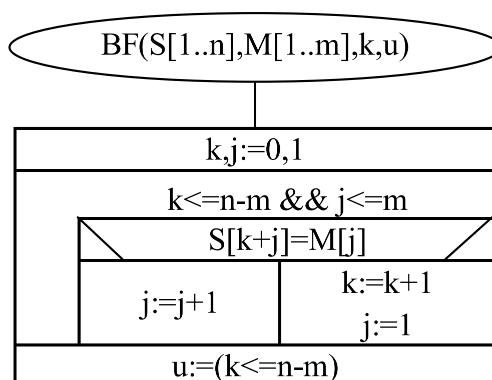
A mintaillesztési feladat egy megoldása:
Tekintsük primitív (megengedett) műveletnek az azonos méretű sorozatok egyenlőségének a vizsgálatát, azaz jelen esetben az $S[k+1..k+m]=M[1..m]$ vizsgálatot.
Ekkor a feladat megkeresni az első olyan *k* pozíciót ($k \in N, k \in [0..n-m]$), amelyre igazat ad a fenti vizsgálat. Ezt megtehetjük egy **lineáris kereséssel**.

A fejezett további részeiben is használt paraméterek jelentése legyen: $u = igaz \Leftrightarrow$ ha $\exists k \in N, k \in [0..n-m]$: M a $k+1$-dik pozíción illeszkedik S-re, továbbá $u=igaz$ esetén k visszatérési értéke az első érvényes eltolás.

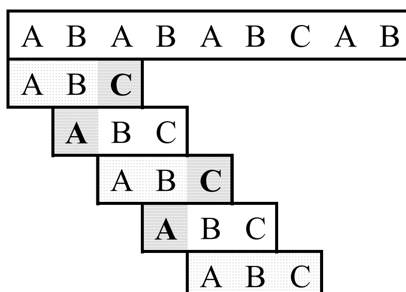


1. Brute-Force algoritmus (BF)

A Brute-Force (nyers erő, egyszerű mintaillesztő) algoritmushoz könnyen eljuthatunk a már tanult programozási tételekre való visszavezetéssel.
Folytassuk az előző részben elkezdett, lineáris keresésre épülő gondolatot. A vázolt megoldásban megengedett műveletnek tekintettük a $S[k+1..k+m]=M[1..m]$ vizsgálatot. Ennek a kifejezésnek az eredményét megkaphatjuk **karakterenkénti összehasonlítással** is, amelynek során a minta minden karakterét összehasonlítjuk a szövegdarab megfelelő karaktereivel, ha az összes vizsgált karakter egyezik, a kifejezés értéke legyen igaz, különben hamis.
A $S[k+1..k+m]=M[1..m]$ vizsgálat előbb említett megvalósítása **javítható**, ha visszavezetjük **lineáris keresésre**, amelynek során keressük az első olyan $j \in N,\ j \in [1..m]$ pozíciót, amelyre $S[k+j] \neq M[j]$. Amennyiben nem találunk ilyen j pozíciót, azaz $\forall j \in N,\ j \in [1..m] : S[k+j] = M[j]$, definíció szerint az M illeszkedik S-re k eltolással, tehát $S[k+1..k+m]=M[1..m]$ vizsgálat eredménye legyen igaz, különben pedig hamis.
Ezt a megoldást nevezzük Brute-Force algoritmusnak, amely nem más, mint egy **lineáris keresésbe ágyazott lineáris keresés**.
Az algoritmust szemléletesen úgy lehet elképzelni, mintha a mintát tartalmazó "sablont" tolnánk végig a szövegen, és balról jobbra ellenőrizzük, hogy a minta karakterei egyeznek-e a "lefedett" szöveg karaktereivel. Amennyiben nem egyező karakterpárt találunk, a mintát eggyel jobbra "toljuk" a szövegen, és a megint kezdjük a minta elejéről az összehasonlítást.



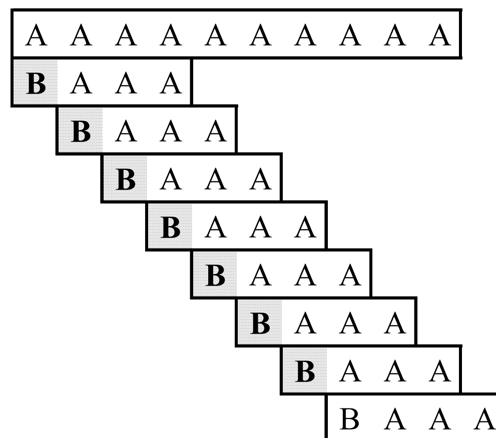
Most nézzük meg **egy példán** az algoritmus működését. Az ábra első sorában szerepel a szöveg, alatta a minta a megfelelő eltolásokkal. A mintán ritka pontozású háttérrel jelöltük, azokat a karaktereket, amelyeknél az összehasonlítás igaz értéket adott, és sűrű mintázatú háttérrel, ahol az illeszkedés elromlott.



Először mintát a szöveg első pozíciójára illesztjük, majd a mintán balról jobbra haladva vizsgáljuk a karakterek egyezését a szöveg megfelelő karaktereivel. A minta első illetve második karaktere ('A' és 'B') megegyezik a szöveg első és második karakterével, azonban a minta harmadik karaktere ('C') nem azonos a szöveg harmadik karakterével, tehát a minta nem illeszkedik az első pozícióra. "Toljuk el" a mintát egyel, majd a minta elejétől kezdve balról jobbra haladva ismét vizsgáljuk a karakterek egyezését a lefedett szövegrész megfelelő karaktereivel. Már a minta első karakterénél ('A') elromlik az illeszkedés. Ismét "toljuk el" eggyel jobbra a mintát, és a már ismertetett módon vizsgáljuk az illeszkedést.
Tizenegy összehasonlítás után eljutunk a végeredményhez, amely szerint a minta az 5. pozíción illeszkedik először a szövegre.

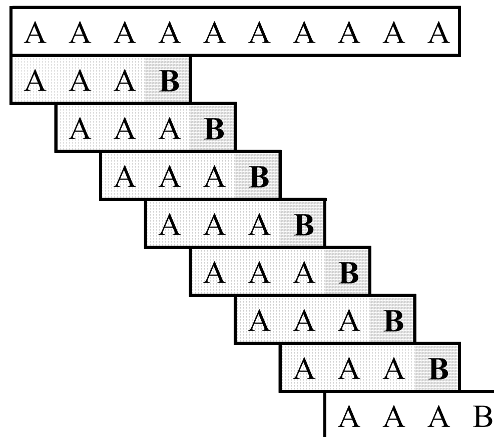
Műveletigény:
Műveletigény szempontjából a legjobb eset, amikor a minta az **első pozíción illeszkedik** a szövegre, ekkor az összehasonlítások száma **minden mintaillesztő algoritmusnál** m lesz. Tehát nem szerencsés ezt az esetet legjobb esetként tekinteni, mivel **nem ad** az algoritmus sebességére vonatkozóan **valósághű jellemzést**. **Továbbiakban** a mintaillesztési algoritmusok vizsgálata során **mindig tegyük fel, hogy a minta nem fordul elő a szövegben**, így az algoritmusnak fel kell dolgoznia a "teljes" szöveget (a szöveg legalább akkora részét, amelyből már következik, hogy a minta nem fordul elő a szövegben). A különböző algoritmusok hatékonysága abban fog különbözni, hogy mennyire "gyorsan" tudnak "végig menni" a szövegen.
Tegyük fel még azt is, hogy a minta hossza nagyságrendben nem nagyobb, mint a szöveg hossza, azaz $m = O(n)$ (a gyakorlatban a minta hossza jóval kisebb, mint a szöveg hossza).

A Brute-Force algoritmus műveletigénye:
Legjobb eset: A minta első karaktere nincs a szövegben, így minden k eltolásnál már $j=1$ esetben mindig elromlik az illeszkedés. Tehát minden eltolásnál csak egy összehasonlítás van, azaz az összehasonlítások száma **megegyezik az eltolások számával**, n-m+1 -gyel.
Pl.:



Tehát $mÖ(n,m) = n - m + 1 = \Theta(n)$.

Legrosszabb eset: A minta minden eltolásánál csak a minta utolsó karakterénél romlik el az illeszkedés. Ekkor minden eltolásnál m összehasonlítást végzünk, így a műveletigény az **eltolások számának *m* szerese**.
Pl.:

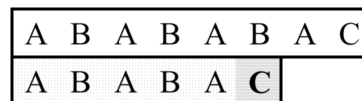


Tehát $MÖ(n,m) = (n-m+1) * m = \Theta(n * m)$.

Szekvenciális sorozatokra, fájlokra való alkalmazhatóság:
A gyakorlatban igen gyakran az általunk szövegnek nevezett sorozat, nagy méretű, emiatt **szekvenciálisan elérhető formában** áll rendelkezésünkre, amelyen **nem megengedett művelet az indexelés**. Nem haszontalan annak vizsgálata, hogy az ismertetett algoritmust milyen egyszerű átírni szekvenciális sorozatokra, fájlokra. A Brute-Force algoritmus szekvenciális sorozatokra való átírásánál kénytelenek vagyunk **buffert használni**, mivel a szövegben időnként (ha az illeszkedés nem a minta első karakterénél romlik el) vissza kell "ugrani".
(Részletesebben lásd gyakorlaton.)

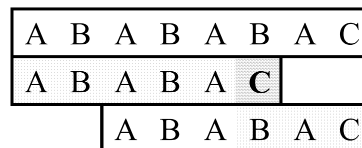
2. Knuth-Morris-Pratt algoritmus (KMP)

A Brute-Force algoritmus műveletigénye legrosszabb esetben $m*n$-es volt, próbáljuk ezen javítani. Amikor az illeszkedés elromlott, a mintát eggyel eltoltuk és a minta elejétől újra kezdtük a minta és a lefedett szövegrész összehasonlítását. Azonban a már vizsgált szövegrészt nem biztos, hogy újra meg kell vizsgálni. Amennyiben az illeszkedés elromlik, akkor egy hibás kezdetünk van, de ez a kezdet ismert, mivel az elromlás előtti karakterig egyezett a mintával. Ezt az információt használjuk fel, hogy elkerüljük az állandó visszalépést a szövegben a minta kezdetére (ill. a kezdete utáni pozícióra).
Pl.:



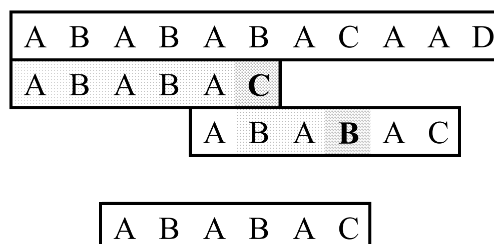
A fenti példában a minta 6. pozíciójánál romlik el az illeszkedés ⇒ a minta első 5 pozíciója illeszkedett. Kérdés hova pozícionáljuk a mintát a szövegbe, és honnan vizsgáljuk tovább az illeszkedést, hogy a minta előfordulását megtaláljuk (ha létezik, át ne "ugorjuk") és az eddig megszerzett 5 illeszkedő karakternyi információt felhasználjuk.

Ha a vizsgálatot az alábbi módon folytathatnánk:



Látható, hogy a minta illeszkedő részének ($M[1..5]$) van egy olyan valódi kezdőszelete (valódi **prefix**e), amely egyezik ezen illeszkedő rész egy valódi végszeletével (valódi **szuffix**ével), azaz $M[1..3]=M[3..5]$ ('ABA'='ABA').
Egy prefix vagy szuffix valódi ,ha hossza legalább 1, és kisebb mint annak a sorozatnak a hossza, amelynek a prefixe vagy szuffixe.
Amennyiben a mintával akkorát ugrunk, hogy a minta eleje az említett szuffixnél kezdődjön, azaz az említett prefix a szuffixel kerüljön fedésbe, a prefixet már nem kell újra vizsgálni, mivel az azonos a szuffixel, ami megegyezik a szöveg lefedett részével, mivel az részsorozata az eredetileg illeszkedő $M[1..5]=S[k+1..k+5]$ szövegrésznek. Ezek után az illeszkedés vizsgálatot a szöveg "elromlott" $S[k+6]$ karakterével, és az említett prefix utáni első karakterrel lehet tovább folytatni.

Mi van akkor, ha több ilyen egyező prefix és szuffix pár is van? A példában is találhatunk egy másik párost, az $M[1..1]=M[5..5]$ ('A'='A').



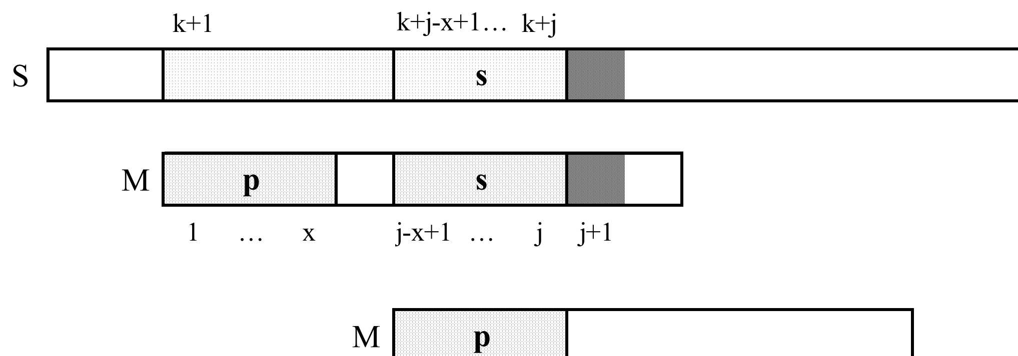
A	B	A	B	A	C

Ha ennek megfelelően pozícionáljuk a mintát, majd a következő karaktertől kezdünk összehasonlítani, azt tapasztaljuk, hogy nem illeszkedik a minta a szövegre, "átugrottunk" egy illeszkedést.

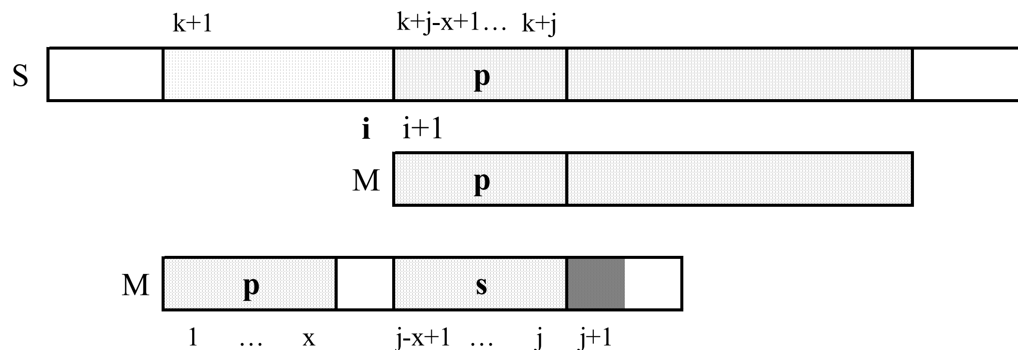
Tehát a legkisebb olyan ugrást kell választanunk, ahol a minta $M[1..5]$ részsorozatának egy prefixe illeszkedik a részsorozat egy szufixére. Akkor "ugrunk" a legkisebbet, ha a legnagyobb ilyen prefixet választjuk.

Az **algoritmus helyességének az igazolásához** az alábbi kérdéseket kell tisztáznunk:
Tegyük fel, hogy a minta $M[1..j]$ részsorozata illeszkedett a szöveg $S[k+1..k+j]$ részsorozatára és az illeszkedés a következő pozíción romlott el, azaz $M[1..j]=S[k+1..k+j]$ és $M[j+1]\neq S[k+j+1]$.

1. Ha létezik $M[1..j]$-nek létezik olyan valódi prefixe ($p=M[1..x]$) és szuffixe ($s=M[j-x+1..j]$), hogy $p=s$, akkor **az ugrás után biztosan nem kell újra vizsgálni** az $M[1..x]$ és az általa lefedett $S[k+j-x+1..k+j]$ szövegrészt?
   Biztosan nem kell, mivel $p=s$, azaz $M[1..x]=M[j-x+1..j]$, továbbá az illeszkedés az $M[j+1]$ pozíción romlott el, tehát $M[1..j]=S[k+1..k+j]$, és ezek tetszőleges, jelenleg fedésben lévő részsorozatai is azonosak, azaz $S[k+j-x+1..k+j]=M[j-x+1..j]$ $\Rightarrow$ $M[1..x]=S[k+j-x+1..k+j]$.



2. Mit tegyünk, **ha nincs ilyen egymással megegyező valódi prefix-szuffix páros**?
   Mivel $M[1..j]$ illeszkedett és $M[j+1] \neq S[k+j+1]$, $\forall i\,(k < i < k+j)$ eltolásra a minta biztosan nem fog illeszkedni, mivel az alábbi sematikus ábrán is látszik, ahhoz hogy ilyen i eltolással illeszkedjen, az kellene, hogy legyen legalább 1 hosszú valódi, egymással azonos prefix-szuffix páros ($p = s$), mert az $M[1..x] = S[k+j-x+1..k+j]$ részsorozatoknak illeszkedniük kell (ekkor $i = k+j-x$) ahhoz, hogy teljes illeszkedés lehessen. Ebből pedig következik, hogy $M[1..x] = M[j-x+1..j]$, mivel $M[1..j] = S[k+1..k+j]$. (Tehát beláttuk az $M[1..j] = S[k+1..k+j]$ feltétel esetén, az $i\ (k < i < k+j)$ **érvényes eltolás szükséges feltételét** is.)



   Tehát a mintával "átugorhatjuk" a már vizsgált $S[k+1..k+j]$ részt, és az illesztést a minta elejétől és a szöveg $S[k+j+1]$ pozíciójától újra kezdhetjük.

3. Mit tegyünk, **ha több ilyen egymással megegyező, valódi prefix-szuffix páros is van**?
   Ha több ilyen prefix-szuffix páros is van, akkor a **leghosszabbat kell venni**, mert ekkor "ugrunk" a legkisebbet. Ilyenkor nem fordulhat elő, hogy átugrunk egy előfordulást, mivel a leghosszabbnál hosszabb, ilyen egymással megegyező, valódi prefix-szuffix páros nincs, ami pedig az $i\ (k < i < k+j)$ érvényes eltolás szükséges feltétele lenne.

Definiáljuk a ***next*** **függvényt**, amely megadja a minta egyes kezdőrészleteire a leghosszabb egymással egyező prefix-szuffix párok hosszát. Ezt felhasználva tudjuk megadni a mintával való "ugrás" mértékét.

Definíció:

$\forall j \in [1..m-1]$ egészre : $\quad next(j) := max\{h \in [0..j-1]$ egész, ahol $M[1..h] = M[j-h+1..j]\}$

Megjegyzések:

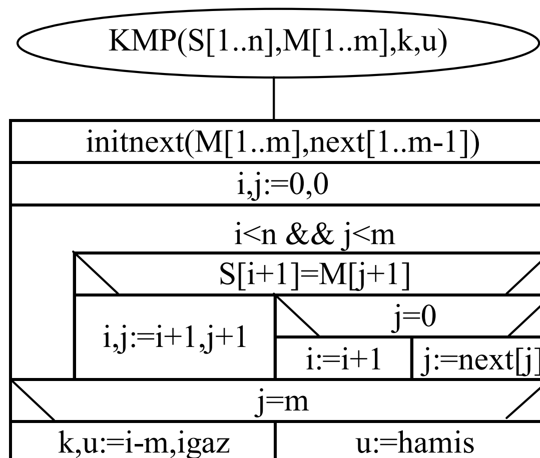
- A $next$ értelmezési tartományát elegendő $m-1$-ig definiálni, mivel ha $j = m$-ig illeszkedik a minta, akkor találtunk egy érvényes eltolást, tehát készen vagyunk, és nem kell a mintával tovább "ugrálnunk".
- A h legkisebb 0 értékét, akkor veszi fel a függvény, ha nincs a minta $M[1..j]$ kezdőszeletében egymással megegyező, valódi prefix-szuffix páros. Továbbá, ha létezik ilyen prefix-szuffix páros, az attól lesz valódi, hogy a hosszát $j-1$-el felülről korlátozzuk.
- A ***next*** **függvény csak a mintától függ**, így értékeit a minta ismeretében a keresés előtt kiszámíthatjuk, és eltárolhatjuk egy $next[1..m-1]$ vektorban.

Most nézzük az algoritmus működését:

A minta elejétől elkezdjük összehasonlítani a szöveg és a minta egymással fedésben lévő karaktereit. Amennyiben a szöveg és minta karakterei azonosak, akkor mind a szövegben, mind a mintában eggyel továbblépünk Azonban, ha a karakterek különböznek, a következőket tesszük:

- Ha a minta elején állunk ($j = 0$ esetén): a szöveg következő pozíciójától ($S[k+j+2]$) és a minta elejétől kezdve újra kezdjük az illeszkedés vizsgálatot, mivel a next függvény a valódi prefix-szuffix hosszát adja meg, de az 1 hosszú sorozatnak nincs valódi prefixe vagy szuffixe ($next[1] = 0$).
- Ha nem a minta elején állunk ($j > 0$ esetén): most jöhet a *next* függvénynek megfelelő "ugrás", amelyet úgy valósítunk meg, hogy azt mondjuk, hogy eddig *j* hosszon illeszkedett a minta, továbbiakban $next[j]$ hosszon illeszkedik. Az összehasonlítást a minta $M[next[j]+1]$ karakterétől és a szöveg $S[k+j+1]$ karakterétől folytatjuk, azaz a szövegben onnan, ahol az illeszkedés elromlott.

Mivel a szövegben legfeljebb 1 hosszú lépésekkel haladunk végig, az egyszerűség kedvéért a *k* eltolásnak megfelelő változó helyett használjunk egy *i* nevű változót, amellyel a szövegben szekvenciálisan haladunk ($i = k + j$), majd az algoritmus végén beállítjuk a *k* változó értékét.



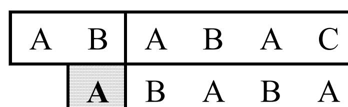
A *next* vektor értékeinek a kiszámítása:

Az *initnext* eljárás során töltjük fel a *next* vektort. A feltöltés ötlete, **a minta elcsúsztatott keresése önmagán** (KMP algoritmussal), miközben feljegyezzük a **legnagyobb illeszkedő részek** hosszát.

Nézzük meg egy példán a feltöltés menetét. Legyen a minta M='ABABAC'.

Már korábban megbeszéltük, hogy $next[1] = 0$.

Ezután a $next[2]$ értékét szeretnénk megkapni. Ekkor az $M[1..2]$ kezdőrészletnek keressük a legnagyobb egymással megegyező, valódi prefix-szuffix párját. A legnagyobb ilyen valódi prefix-szuffix 1 hosszú lehet. Tehát az a kérdés, hogy az $M[1] = M[2]$ egyenlőség teljesül-e? Ehhez a mintát csúsztassuk el eggyel, és a fedésben lévő karaktereket vizsgáljuk:



Látható, hogy nem azonosak, így $next[2] = 0$.

Most a $next[3]$ meghatározása a cél. Ekkor az $M[1..3]$ kezdőrészletnek keressük a legnagyobb egymással megegyező, valódi prefix-szuffix párját. A legnagyobb ilyen valódi prefix-szuffix 2 hosszú lehet. Azaz $M[1..2] = M[2..3]$ egyenlőség teljesül-e? Azonban ez nem teljesülhet, mivel már $M[1] = M[2]$ sem teljesült. Ezt nem is vizsgáljuk, mivel már az előző menetben sem volt egyezés. Helyette a mintát eggyel jobbra csúsztatjuk, és az $M[1] = M[3]$ egyenlőséget vizsgáljuk:

A	B	A	B	A	C
		A	B	A	B

,amely teljesül, ezért feljegyezzük $next[3] = 1$.

Most a $next[4]$ meghatározása a cél. Ekkor az $M[1..4]$ kezdőrészletnek keressük a legnagyobb egymással megegyező, valódi prefix-szuffix párját. A legnagyobb ilyen valódi prefix-szuffix 3 hosszú lehetne, de $M[1] = M[2]$ egyenlőséget már korábban is megvizsgáltuk és nem teljesült, így ez szóba sem jöhet. Azonban az előző menetben $M[1] = M[3]$ teljesült, így az ennek megfelelő elcsúsztatott pozíciót megtartva vizsgáljunk, mert további karakter egyezés esetén ez lehetne a leghosszabb prefix-szuffix pár:

A	B	A	B	A	C
		A	B	A	B

Valóban az $M[2] = M[4]$ teljesül, így feljegyezzük $next[4] = 2$.

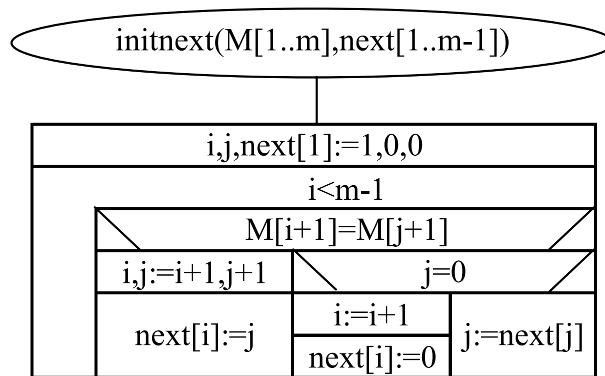
A $next[5]$ meghatározásához, az előző menethez hasonlóan a mintát nem csúcstatjuk el, hanem a következő karaktert vizsgáljuk:

A	B	A	B	A	C
		A	B	A	B

Azt látjuk, hogy $M[3] = M[5]$, így feljegyezzük $next[5] = 3$.

Összefoglalva:

j	next[j]	A	B	A	B	A	C
1	0						
2	0		A	B	A	B	A
3	1			A	B	A	B
4	2			A	B	A	B
5	3			A	B	A	B



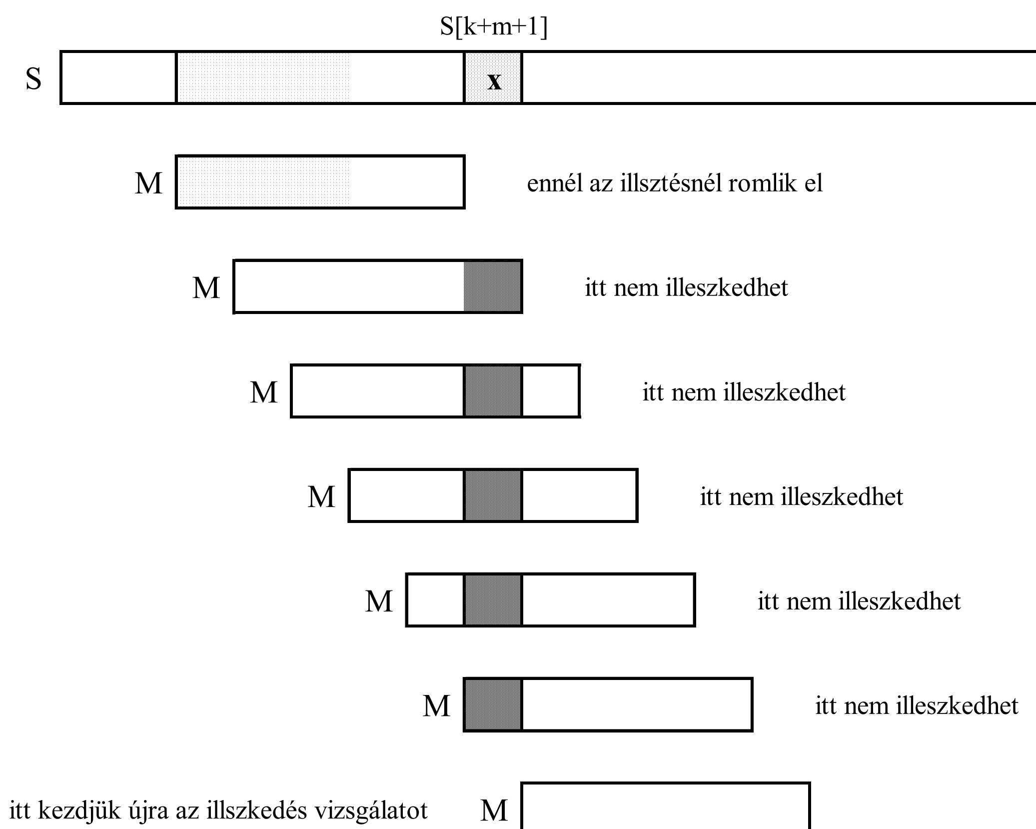
Műveletigény:
Az *initnext* műveletigénye $\Theta(m)$. Tegyük fel, hogy $m << n$, ekkor a keresés műveletigénye legjobb és legrosszabb esetben is $\Theta(n)$. (A legjobb és legrosszabb eset megegyezik a BF algoritmus legjobb és legrosszabb esetével. Nem kell levezetni.) Tehát a KMP algoritmus stabil algoritmus.

Szekvenciális sorozatokra, fájlokra való alkalmazhatóság:
Mivel a KMP algoritmus működése során a szövegben csak legfeljebb egy pozícióval történő előre lépést teszünk (nincs visszalépés), így az algoritmus buffer használata nélkül is átírható szekvenciális sorozat/fájl formában adott szövegre.

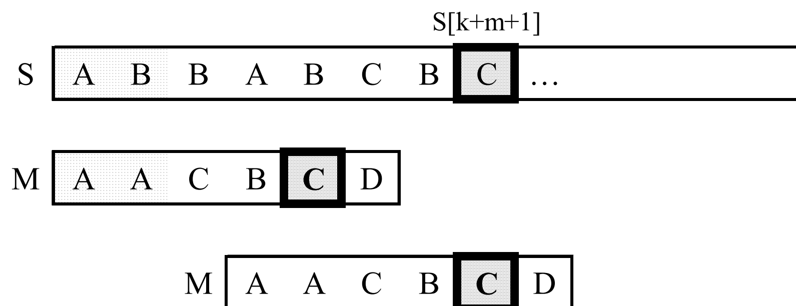
3. Quick-Search algoritmus (QS)

A Brute-Force algoritmus műveletigénye legjobb esetben n-es volt, most ezen próbálunk javítani. A QS algoritmus esetén is a mintát és szöveget balról jobbra hasonlítjuk össze. Ha az illeszkedés elromlik vizsgáljuk meg **a szöveg minta után eső részének első karakterét** ($S[k+m+1]$). Ennek a karakternek a függvényében fogjuk eldönteni, hogy mekkorát "ugorjunk" a mintával. Az alábbi két esetet különböztetjük meg:

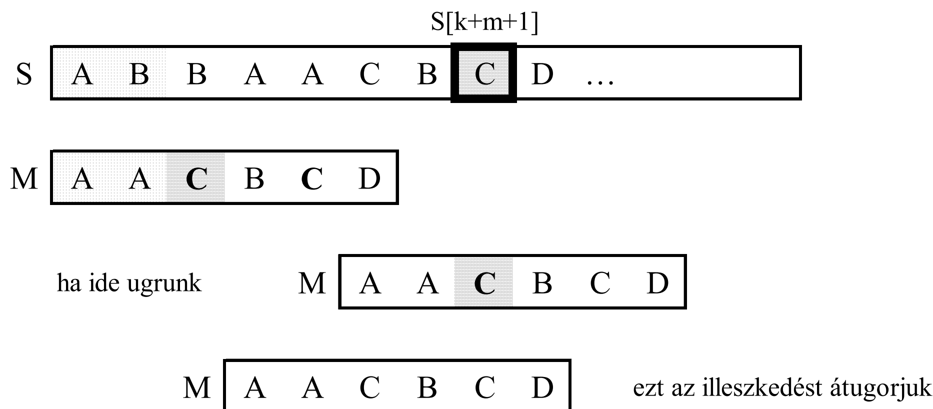
1. **A szöveg minta utáni első karaktere nem fordul elő a mintában**, azaz $S[k+m+1] \notin M$. Ekkor bármely olyan illesztés sikertelen lenne, ahol az $S[k+m+1]$ pozíció fedésbe lenne a mintával. Tehát ezt a pozíciót "átugorhatjuk" a mintával, és a szöveg következő ($S[k+m+2]$) karakterétől kezdhetjük újra (a minta elejétől) az illeszkedés vizsgálatot.



2. **A szöveg minta utáni első karaktere előfordul a mintában**, azaz $S[k+m+1] \in M$. Ekkor vegyük a mintabeli előfordulások közül "jobbról" az elsőt, és akkorát "ugorjunk" a mintával, hogy az említett mintabeli karakter, fedésbe kerüljön a szöveg $S[k+m+1]$ karakterével, majd a minta elejétől kezdve újra vizsgáljuk az illeszkedést. Mivel úgy találtuk, hogy k nem lehet érvényes eltolás, egy i ($k < i < k+m+1$) eltolás akkor lehet érvényes, ha $S[k+m+1]$ karakter fedésbe kerül a minta egy azonos karakterével.



Azért az említett i eltolásokat tekintjük, mivel ekkor kerülhet fedésbe $S[k+m+1]$-gyel valamely mintabeli karakter. Továbbá $S[k+m+1]$ karakternek azért jobbról az első mintabeli előfordulását tekintjük, mert ekkor "ugrunk" a legkisebbet, tehát nem kell attól tartani, hogy egy jó illeszkedést "átugrunk", mint az alábbi példában.



A mintával való "ugrás" végrehajtásához bevezetjük a *shift* függvényt. A *shift* függvény az ABC minden betűjére megadja az "ugrás" nagyságát, amelyet akkor tehetünk, ha az illeszkedés elromlása esetén az illető betű lenne a szöveg minta utáni első karaktere.

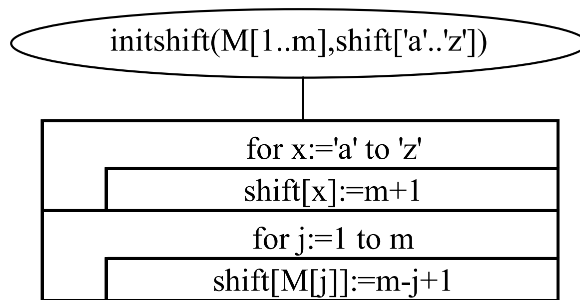
Definíció: $shift : H \rightarrow [1..m+1]$

$$shift(x) = \begin{cases} m+1 & ,ha\ x \notin M \\ m-j+1 & ,ha\ M[j] = x \wedge \forall i \in [j+1..m] : M[i] \neq x \end{cases}$$

Megjegyzések:

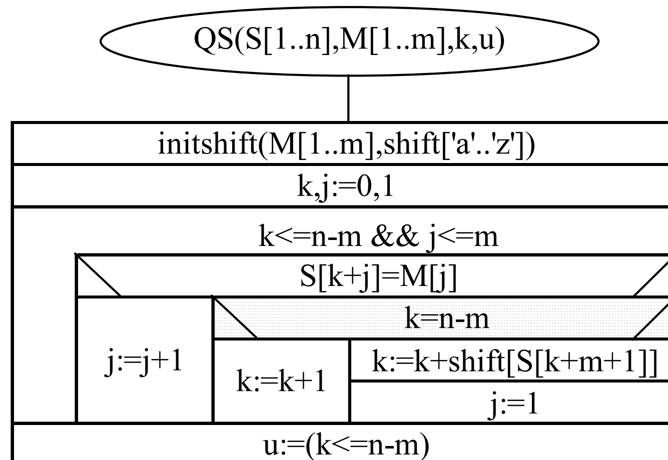
- A definícióban is jól láthatóan elválik a fent említett két eset.
- Mivel a *shift* függvény csak a mintától függ, értékeit előre kiszámíthatjuk, és eltárolhatjuk egy vektorba, amit az ABC betűivel indexelünk.

Most nézzük a ***shift* függvény** értékeinek a kiszámítására szolgáló ***initshift*** eljárást:



A első ciklus feltölti a *shift* tömböt *m+1*-el. A második ciklus adja meg egy betű, jobbról az első mintabeli elfordulásának megfelelő ugrás nagyságát. Azért balról-jobbra megyünk végig a mintán, hogy a többször előforduló betűk esetén az utolsó (jobbról az első) előfordulásnak megfelelő "ugrást" jegyezzük fel.

A Quick-Search algoritmus struktogrammja:

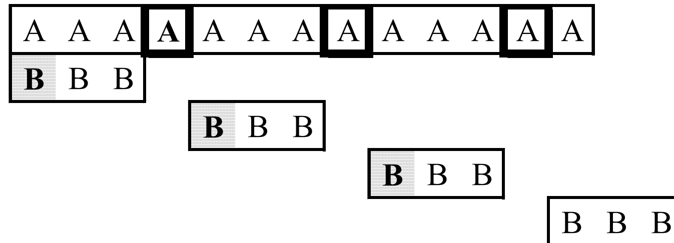


Az algoritmus működése nagyon egyszerű. Először kiszámítjuk a *shift* függvény értékeit, majd *k* eltolásokkal próbáljuk illeszteni a mintát. Amennyiben az illeszkedés elromlik, a *shift* függvénynek megfelelő ugrással változtatjuk az eltolás mértékét. A pontozott hátérrel kiemelt elágazással, ügyelünk arra, hogy amikor a mintát a szöveg végére illesztjük, és az illeszkedés elromlik, akkor ne olvassunk túl a szövegen, mivel ekkor $S[k+m+1] = S[n-m+m+1] = S[n+1]$.

Műveletigény:

Legjobb eset: A minta olyan karakterekből áll, amelyek nem fordulnak elő a szövegben, így a minta első karakterénél már elromlik az illeszkedés, továbbá a minta utáni karakter sem fordul elő a mintában, így azt "átugorhatjuk".

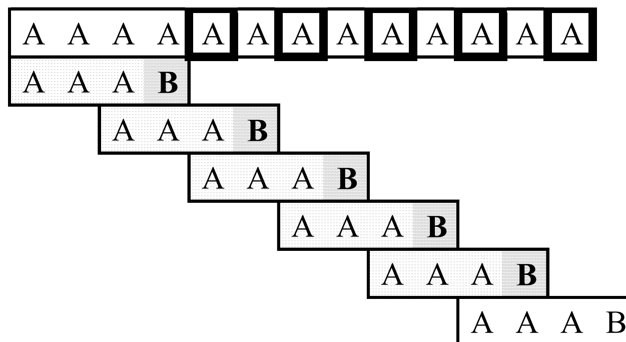
Pl:



$$mÖ(n,m) = \Theta\left(\frac{n}{m+1}\right)$$

Legrosszabb eset: A minta végén romlik el az illeszkedés és csak kicsiket tudunk "ugrani".

Pl:



$$MÖ(n) = \Theta(n * m)$$

Szekvenciális sorozatokra, fájlokra való alkalmazhatóság:

Csak buffer használatával lehet alkalmazni, mivel a legrosszabb esetben is láttuk, hogy szükség volt a szövegben való visszalépésre.

4. Rabin-Karp algoritmus (RK)

Ötlet: Egy egész számokból álló sorozaton lineáris kereséssel keressünk egy számot, amelytől azt várjuk, hogy "hatékony" lesz, mivel az egész számokkal való műveletek gyorsan végrehajthatóak.

Legyen az ABC a $H = \{'A','B','C',...,'Z'\}$ véges halmaz, melynek **elemeit sorszámozzuk** meg $[0..d-1]$ közötti egész számokkal. Továbbá legyen $d = |H|$ a H halmaz számossága. Ekkor egy H feletti **szót** (karakter sorozatot) úgy is tekinthetünk, mint egy ***d*** **alapú számrendszerben felírt számot**. Pl.: "BBAC" szónak megfelelő szám tízes számrendszerben 1102, ha a sorszámok 'A': 0, 'B': 1, 'C': 2.

Vezessük be a következő függvényt, amely megadja egy betű *H*-beli sorszámát: $ord : H \rightarrow [0..d-1]$

Ekkor az $M[1..m]$ mintának megfelelő szám

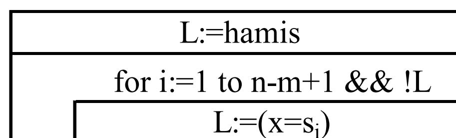
$$x = \sum_{j=1}^{m} ord(M[j])d^{m-j} = ord(M[1]).d^{m-1} + ord(M[2]).d^{m-2} + ... + ord(M[m])d^{0}$$

,amelyet **Horner algoritmussal** hatékonyabban is kiszámíthatunk

$$x = (...((ord(M[1]) * d + ord(M[2])) * d + ord(M[3])) * d + ... + ord(M[m-1])) * d + ord(M[m])$$

A Rabin-Karp algoritmus lényege, hogy a fent bemutatott módon kapott *x* számot hasonlítjuk össze minden egyes *k* eltolás esetén, a szöveg $S[k+1..k+m]$ részsorozatával, mint egy *d* alapú számrendszerben felírt számmal.

Jelöljük s_i-vel az $S[i..i+m-1]$ szónak megfelelő számot ($i = k+1$). Ekkor a feladatot **visszavezethetjük egy egész számok sorozatán való lineáris keresésre**:



Pl.: Legyen a szöveg S="DACABBAC", a minta M="BBAC".

	D	A	C	A	B	B	A	C
s_1	3	0	2	0				
s_2		0	2	0	1			
s_3			2	0	1	1		
s_4				0	1	1	0	
s_5					1	1	0	2

x	1	1	0	2

A példában is látható, hogy x-et csak egyszer kell kiszámítani, mivel nem változik, azonban s_i-t minden léptetés után újra kell számolni, ami igen költséges még akkor is, ha Horner algoritmussal számítjuk. **Hogyan lehetne s_{i+1}-et hatékony számolni s_i ismeretében?**
Pl.: legyen 23456 sorozat, aminél a 2345 szám nem egyezik a mintával. A 2345 szám ismeretében állítsuk elő a 3456 számot! A 2345-ből kiléptetjük a "bal szélső" 2-es számjegyet, majd a megmaradt 345 szám számjegyeit egy helyi értékkel balra léptetjük, végül hozzáadjuk a 6-ot, azaz (2345-2*1000)*10+6

Általánosan: $s_{i+1} = \left(s_i - ord(S[i]) * d^{m-1}\right) * d + ord(S[i+m])$

A formula 2 szorzást tartalmaz, mivel d^{m-1}-et előre kiszámíthatjuk, és egy változóban tárolhatjuk, így menet közben nem kell hatványozni.

Elméletileg kész is vagyunk, mivel egy string kölcsönösen egyértelműen megfeleltethető egy egész számnak, így a feladatot visszavezethetjük egy lineáris keresésre, amelynek **műveletigénye** a fent leírt formula használatával **lineáris marad**. A **gyakorlatban** azonban hosszabb minta vagy túl nagy ABC esetén előfordulhat, hogy egy m számjegyből álló, d alapú számrendszerbeli szám **nem tárolható egy egész számként** (túlcsordul). Amennyiben nem egész számtípusként tároljuk, a műveletek ideje megnő. A megoldás, **vegyünk egy kellően nagy *p* prímet** ($d*p$ még ábrázolható legyen) és ***modulo p* számoljuk** x-et és s_i-t. Ekkor az egyértelműséget elveszítjük, mivel **maradékosztályokhoz való tartozást** vizsgálunk, de

1. $x \bmod(p) \neq s_i \bmod(p) \Rightarrow x \neq s_i$
2. $x \bmod(p) = s_i \bmod(p) \Rightarrow$ x és s_i azonos maradékosztályba esik, tehát további vizsgálat szükséges, azaz **karakterenkénti összehasonlítással eldöntjük**, hogy $M[1..m] = S[i..i+m-1]$ egyenlőség teljesül-e.

Hamis találatnak szokás nevezni azt az esetet, amikor $x \bmod(p) = s_i \bmod(p)$, de $M[1..m] \neq S[i..i+m-1]$.
Minél nagyobb p-t választunk, a maradékosztályokba annál kevesebb elem esik, így várhatóan a hamis találatok száma is kevesebb lesz.

Tehát a formula: $s_{i+1} = ((s_i - ord(S[i]) * d^{m-1}) * d + ord(S[i+m])) \bmod(p)$

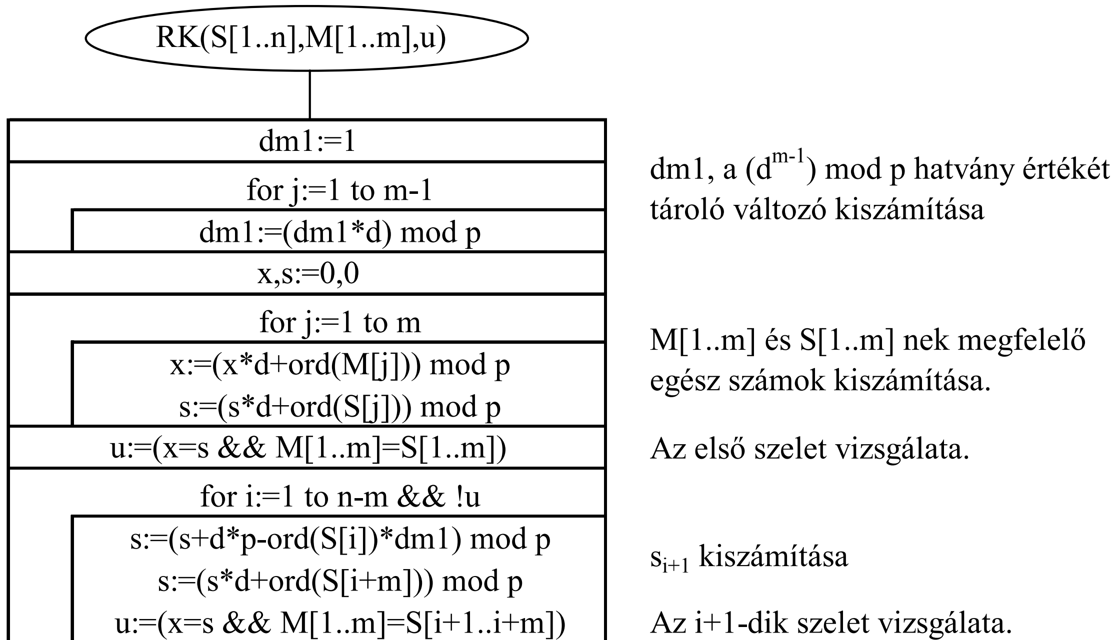
A *modulo* számítás következtében **újabb probléma** merült fel: $s_i - ord(S[i]) * d^{m-1}$ értéke **lehet negatív** is, amely megkövetelné az előjeles egész számok használatát, így viszont abszolút értékben kisebb számok ábrázolhatók, miközben p-t szeretnénk minél nagyobbnak választani. Adjunk s_i-hez ***d*p***-t, ami biztosítja, hogy a különbség nem lesz negatív, és az osztási maradékot sem változtatja.

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

A kifejezés **túlcsordulásának a megelőzésére** számítsuk a kifejezést két lépésben (*modulo* esetén megtehetjük):

$$\begin{cases} s = (s_i + d * p - ord(S[i]) * d^{m-1}) \bmod(p) \\ s_{i+1} = (s * d + ord(S[i+m])) \bmod(p) \end{cases}$$

Tehát a Rabin-Karp algoritmus struktogrammja:



Tegyük fel, hogy az && művelet rövidzáras, azaz a második feltételt csak akkor értékeli ki, ha az első feltétel igaz.

Műveletigény:
Legjobb esetben: egyszerűen végigolvassuk a szöveget, azaz minden esetben $x \neq s$.
$m\ddot{O}(n,m) = \Theta(n)$
Legrosszabb esetben:

- Ha *p* olyan, hogy mindig hamis találatot kapunk, akkor minden esetben karakterenként is megvizsgáljuk az $M[1..m] = S[i+1..i+m]$ egyenlőséget, azaz visszakapjuk a Brute-Force algoritmust. Tehát $M\ddot{O}(n,m) = \Theta(n*m)$.
- Jó *p* esetén, nincs vagy csak nagyon kevés a hamis találat, így $M\ddot{O}(n,m) = \Theta(n)$, tehát egy stabil algoritmus.

Szekvenciális sorozatokra, fájlokra való alkalmazhatóság:
Csak buffer használatával lehet alkalmazni, mivel $x = s$ találat esetén, a szövegben vissza kell lépni, és karakterenként meg kell vizsgálni az $M[1..m] = S[i+1..i+m]$ egyenlőséget.

Felhasznált irodalom

1. Sike Sándor: *Programozási módszertan II: Algoritmusok.* Elektronikus jegyzet
2. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest: *Algoritmusok.* Műszaki Könyvkiadó, 1997, 1999