# String keresések

## Bevezetés

Stringeknek nevezzük a karakterekből álló sorozatokat. Az egyik alapvető feladat amit stringekkel kell végezni az úgynevezett *mintaillesztés*. Ennek lényege, hogy egy *N* hosszúságú stringben - *szövegben* - keressük meg egy *M* hosszúságú string - *minta* - egy előfordulását. Általában az első előfordulás megtalálása a feladat, de gyakori probléma az összes előfordulás meghatározása is. A továbbiakban az első előfordulás megtalálásával foglalkozunk, hiszen ennek a feladatnak a megoldása könnyen módosítható, úgy hogy a második problémára is megoldást kapjunk. Szinte minden esetben igaz, hogy a minta hossza lényegesen kisebb mint a szöveg hossza, azaz $M \ll N$. (Egy ilyen alkalmazás például szövegszerkesztő programok esetén egy szó keresése a szövegben.)

A stringeket a továbbiakban *vektor*okkal fogjuk ábrázolni. Ebben az esetben ez nem jelent megszorítást a sorozathoz képest, hiszen az indexelés ugyanúgy értelmezett mindkét esetben, és a sorozatok hossza most nyilvánvalóan nem változik. (Ez természetesen nem vonatkozik arra az esetre ha az indexelés nem megengedett mint pl. szekvenciális fájlok esetén.) A vektorok értékei egy véges halmazból kerülnek ki, amit *ábécé*nek nevezünk.

Legyen *H* az ábécé, és a szövegnek illetve a mintának megfelelő típusok az alábbiak:

$$S = \text{vekt}\,(\,[1..N],\ H\,)$$
$$M = \text{vekt}\,(\,[1..M],\ H\,)$$

Ezek után a feladat formális specifikációja (*L* a logikai értékek, *N* pedig a természetes számok halmazát jelöli):

$$A = \underset{s}{S} \times \underset{m}{M} \times \underset{u}{L} \times \underset{i}{N}$$

$$B = \underset{s'}{S} \times \underset{m'}{M}$$

$$Q = (\, s = s' \wedge m = m' \,)$$

$$R = Q \wedge u = (\exists j \in [1..N-M+1]: \mathit{match}(j,M)) \wedge$$
$$u \Rightarrow (i \in [1..N-M+1] \wedge \mathit{match}(i,M) \wedge \forall j \in [1..i-1]: \neg \mathit{match}(j,M))$$

ahol a match függvény a következő:

$$\text{match}: [1..N-M+1] \times [0..M] \to L, \text{ és}$$
$$\text{match}(i,h) = \forall k \in [0..h-1]: s[i+k] = m[1+k],$$

azaz a match(i,h) függvény értéke akkor és csak akkor igaz, ha a szövegben a minta az i. pozíciótól kezdődően h hosszon illeszkedik.

A feladat megoldása visszavezethető lineáris keresésre, ahol a match függvény értékének a kiszámítása egy újabb lineáris keresés. A továbbiakban néhány más megoldási lehetőséget ismertetünk.

## Brute-Force algoritmus

Ez az algoritmus, mint a neve is mutatja (szó szerint nyers erő, vagyis "józan paraszti ész"), egy olyan megoldás ami semmi különlegességet sem tartalmaz, bárki kitalálhatja nagyobb megerőltetés nélkül. Lényegében nem is különbözik a fent vázolt megoldástól, csupán a match függvény kiszámításához nem használ egy külön lineáris keresést, hanem ezt a "fő" lineáris keresésben végzi el egy változó segítségével.

A megoldás alapelve, hogy a szövegben minden egyes pozíción ellenőrizzük, hogy a minta soron következő jele illeszthető-e. Ha igen, akkor mind a mintában mind a szövegben továbblépünk a következő jelre, ha nem akkor a szövegben visszalépünk a minta mostani kezdőpontjára illetve a minta elejére és újra kezdjük az illesztést.

A program egy szekvencia lesz, aminek az első tagja egy ciklus, a második tagja pedig egy elágazás lesz. Annak érdekében, hogy a programot levezethessük vezessük be közbülső állításként az R utófeltétel módosítását, $R^*$-ot:

$$R^* = Q \wedge i \in [1..N] \wedge j \in [0..M] \wedge \mathit{match}^*(i,j) \wedge (i = N \vee j = M) \wedge \\ \forall k \in [1..i-1]: \neg \mathit{match}^*(k,M))$$

ahol $\text{match}^*$ hasonló, mint match csak az első paraméter nem az egyezés kezdetére, hanem végére mutat, azaz:

$$\text{match}^*(i,h) = \forall k \in [0..h-1]: s[(i-h+1)+k] = m[1+k];$$

Nyilvánvaló, hogy ha sikerül programot írni, ami megoldja a Q és $R^*$ által meghatározott feladatot, akkor $R^*$-ból R-be egy elágazással juthatunk. Ennek a feltételét a $j = M$ adja, és ha ez igaz, akkor az $i,u := i-M+1,\text{true}$ értékadás, egyébként az $u := \text{false}$ értékadás szükséges. A szekvencia első tagját adó ciklus levezethető az alábbi **P** invariáns és **t** terminátor függvény segítségével:

$$P = Q \wedge i \in [0..N] \wedge j \in [0..M] \wedge \mathit{match}^*(i,j) \wedge \\ \forall k \in [1..i-1]: \neg \mathit{match}^*(k,M))$$

$$t := (N-M+1)*M-(i-j)*M-j$$

A levezetést az olvasóra hagyjuk. Ennek alapján a megoldó program:

## Brute-Force

<table>
<tr><td colspan="3">$i,j := 0,0$</td></tr>
<tr><td colspan="3">$i < N \wedge j < M$</td></tr>
<tr><td rowspan="2"></td><td colspan="2">\ $s[i+1] = m[j+1]$ /</td></tr>
<tr><td>$i,j := i+1, j+1$</td><td>$i,j := i-j+1, 0$</td></tr>
<tr><td colspan="3">\ $j = M$ /</td></tr>
<tr><td colspan="2">$i,u := i-M+1, true$</td><td>$u := false$</td></tr>
</table>

Ha a fenti program futási idejét megvizsgáljuk, azonnal kiderül, hogy a legrosszabb esetben a ciklusmag végrehajtásainak a száma $N*M$-mel arányos. (Ennyi jel összehasonlításra van szükség.) Egy ilyen esetre példa, amikor mind a szöveg mind a minta egy jelből áll az utolsó pozícióig, és az utolsó jel egy ettől különböző jel. Pl.: csupa 0 amit egy 1 követ. Nyilvánvaló, hogy a program hasonlóan rossz eredményt ad erősen sok ismétlődést tartalmazó szövegek és minták esetén. Szerencsére ilyen 'degenerált' szövegek és minták normális emberi stringek esetén rendkívül ritkán fordulnak elő, ezért programunk azokra jól alkalmazható és a karakter összehasonlítások száma várhatóan közel lesz az ideálishoz.

A program fenti tulajdonsága szükségessé teszi, hogy jobb megoldások után kutassunk, azokra az esetekre gondolva amikor nagyfokú ismétlődések előfordulhatnak. (Pl.: egy bit sorozatban kell megkeresni egy bit mintát.)

## Knuth-Morris-Pratt algoritmus

A Knuth, Morris és Pratt által kifejlesztett megoldás alapötlete, hogy amikor az illeszkedés elromlik, egy hibás kezdet áll a rendelkezésünkre ugyan, de ez a kezdet számunkra előre ismertnek vehető a további illesztések során, hiszen a minta egy kezdőszeletével egyezik meg; a mintát pedig ismerjük. Ezt az információt használjuk fel, és elkerüljük azt, hogy visszalépjünk a szövegben a minta kezdetére. (Azaz nem csökkentjük az i változó értékét.) Ehelyett a mintát illesztjük újra, megfelelően a szövegre, ami azt jelenti, hogy nem feltétlen a minta elejét helyezzük az aktuális pozíció (i) fölé, hanem esetleg a 'közepét'.

Egy egyszerű példa a fentiekre ha a minta az 10000000, és tegyük fel, hogy van egy hibás kezdetünk ami a $j+1$. helyen romlik el (azaz az első j jel illeszkedik a minta elejére). Mikor ezt a hibát észrevesszük, nem kell a szövegben visszalépnünk, hiszen a megelőző $j-1$ jel egyike sem illeszkedhet a minta első jelére. Azaz ebben az esetben a programban az $i := i-j+1$ értékadás helyettesíthető az $i := i+1$ értékadással.

Természetesen általában ez így nem tehető meg, azaz nem lehet teljesen átugrani a már meglevő kezdetet. Például ha az 10100111 mintát keressük az 1010100111 szövegben. Ekkor ugyanis az első hibát az 5. pozíción érzékeljük, de 'vissza kell lépnünk' a 3. pozícióig, különben nem találnánk meg a mintát. Ennek az esetleges 'visszalépésnek' a mértékét azonban előre meghatározhatjuk, hiszen ez csak a mintától függ. Továbbá ez a 'visszalépés' megvalósítható úgy, hogy a szövegben ténylegesen nem lépünk vissza, hanem azt határozzuk meg, hogy a minta mennyivel előbb

kezdődik a mostani pozíciónál, és a mintát ennyivel eltolva illesztjük újra. Azaz a minta első karaktere helyett az eltolásnak megfelelő karaktert hasonlítjuk össze a szöveg megfelelő jelével.

A *next* függvényt használjuk az eltolás mértékének a meghatározására. Azaz next(j) az a maximális érték ahány jel megegyezik a minta elejétől kezdve és a minta j–next(j). pozíciójától kezdve. Formálisan:

$$\text{next}: [1..M-1] \rightarrow [0..M-1], \text{ és}$$
$$\text{next}(j) = \max(S(j)) \qquad (j \in [1..M-1])$$
ahol:
$$S(j) = \{\, k \in [0..j-1] \mid \forall h \in [0..k-1]: m[1+h] = m[j-k+h] \,\}.$$

A fenti 10100111 mintát használva, a next függvény értékei:

| j | next(j) | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | | | | | | | | |
| 2 | 0 | | 1 | 0 | 1 | 0 | 0 | 1 | 1 |
| 3 | 1 | | | 1 | 0 | 1 | 0 | 0 | 1 |
| 4 | 2 | | | 1 | 0 | 1 | 0 | 0 | 1 |
| 5 | 0 | | | | | 1 | 0 | 1 | 0 |
| 6 | 1 | | | | | | 1 | 0 | 1 |
| 7 | 1 | | | | | | | 1 | 0 |

A next függvény rendelkezik az alábbi tulajdonságokkal, melyek a definíció következményei és igazolásuk az olvasó feladata:

- $\forall j \in [1..M-1]: \text{next}(j) < j$
- $\forall i \in [1..N-M+1]: \forall h \in [1..M-1]: \text{match}(i,h) \wedge \neg\text{match}(i,h+1) \Rightarrow$
  $\text{match}(i+h-\text{next}(h),\text{next}(h)) \wedge$
  $\forall k \in [\text{next}(h)+1..h-1]: \neg\text{match}(i+h-k,k)$

Miután next csak a mintától függ, ezért a függvény helyettesíthető egy vektorral, amelynek értékeit előre kiszámolhatjuk (initnext). Ezek után a megoldó programot a következőképpen kapjuk az előzőből. Csak a ciklusmag változik, mégpedig az az eset amikor a szöveg és a minta vizsgált jele nem egyezik. (Egyezés esetén nyilván az eddigieknek megfelelően mindkettőben tovább kell lépni.) Ha a jelek nem egyeznek és a minta legelején vagyunk, akkor a szövegben tovább kell lépnünk és a minta elején maradni; ha pedig a mintából már egy rész illeszkedett, akkor a mintát újra kell illeszteni a szövegre, azaz j helyett csak az első next(j) jelet tekinteni illesztettnek. Ezek után a program, amelynek formális levezetését - a már ismert R*, P, t segítségével és a fenti tulajdonságokkal - az olvasóra bízunk:

# Knuth-Morris-Pratt

```
initnext
i,j := 0,0
i < N ∧ j < M
    \ s[i+1] = m[j+1] /
        igaz:  i,j := i+1,j+1
        hamis: \ j = 0 /
                   igaz:  i := i+1
                   hamis: j := next[j]
\ j = M /
    igaz:  i,u := i–M+1,true
    hamis: u := false
```

**Könnyen látható, hogy az initnext program nem más mint a minta 'elcsúsztatott keresése' önmagában azzal a kiegészítéssel, hogy a maximálisan illeszkedő szeletek hosszát fel kell jegyeznünk a next vektorban. Ennek alapján nyilvánvaló, hogy a program is lényegében ugyanaz mint a főprogrambeli szekvencia első tagja.** Tehát:

```
initnext
i,j,next[1] := 1,0,0
i < M–1
    \ m[i+1] = m[j+1] /
        igaz:  i,j := i+1,j+1
               next[i] := j
        hamis: \ j = 0 /
                   igaz:  i,next[i+1] := i+1,0
                   hamis: j := next[j]
```

Bebizonyítható, hogy a fenti algoritmusban az összehasonlítások száma a legrosszabb esetben is N+M-mel arányos. (Éppen a Brute-Force algoritmusnál bemutatott 'rossz' eset a legrosszabb, és látható, hogy ekkor az összehasonlítások száma eléri de nem haladja meg a fenti számot, ha a minta nem vagy a szöveg végén szerepel.) Ez lényeges javulás a Brute-Force algoritmushoz képest, és mint látni fogjuk ennél jobbat nem is lehet elérni, azaz minden további algoritmushoz megadható olyan eset amikor az legalább ennyi összehasonlítást igényel. (Ez általában a már bemutatott rossz eset.)

Az algoritmus további előnye, hogy gond nélkül adaptálható szekvenciális fájlokra is. Látható, hogy az algoritmusban i értéke vagy eggyel nő, vagy nem változik. Ez azt jelenti, hogy minden egyes növelés helyettesíthető egy olvasással (az előreolvasással együtt) és máris szekvenciális fájlra cserélhetjük a vektort. A Brute-Force algoritmus esetén ez így nem tehető meg; az csak ún. pufferezett olvasással adaptálható szekvenciális fájlokra. A puffer nagysága a minta nagyságával kell hogy megegyezzen. (Ez egy kivételével a további algoritmusokra is igaz, amit az olvasó könnyen ellenőrizhet majd.)

## Boyer-Moore algoritmus

Ha nem nehéz a szövegben visszafelé haladni, akkor megadható egy olyan algoritmus ami az esetek többségében lényegesen gyorsabban találja meg a keresett mintát, illetve dönti el hogy az szerepel-e a szövegben. Az algoritmust Boyer és Moore fejlesztette ki, és ez a mintát jobbról balra illeszti a szövegre. Az elgondolás alapja, hogy nemcsak a mintát, hanem a nem illeszkedő karaktert is felhasználjuk annak eldöntésére, hogy mit tegyünk a következő lépésben, azaz hol próbáljuk újra illeszteni a mintát a szövegre.

Egy előfeldolgozó lépésben minden lehetséges karakterre megadjuk, hogy az a minta melyik jelére illeszkedik. Pontosabban a minta végéhez közelebbi illeszkedést tartjuk nyilván a mintában előforduló jelekre; illetve azt hogy a jelet átugorhatjuk a mintával ha a jel nem szerepel a mintában. Ennek tárolására bevezetjük a *skip* vektort:

skip = vekt( H, [0..M] ), és
  skip[x] := M , ha $x \notin m$; illetve
  skip[x] := M–j, ahol $m[j] = x$ és $\forall k \in [j+1..M]$: $m[k] \neq x$.

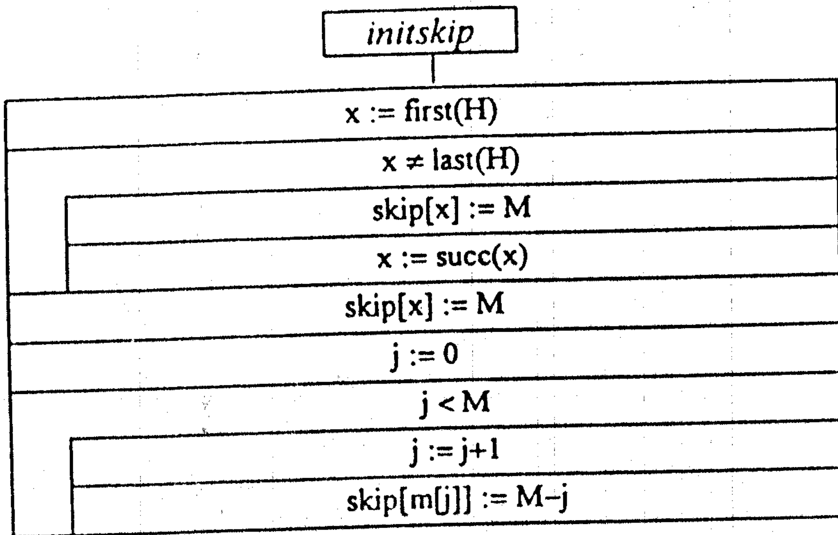
Az algoritmusban a mintát a szöveg elejére illesztjük, majd a minta vége felől ellenőrizzük az illeszkedést. Ha az illeszkedés a szöveg i. karakterén elromlik, a mintát újra illesztjük úgy, hogy ez az egyezés létrejöjjön ezen a pozíción (i:=i+skip[s[i]]), és újra ellenőrizzük az illeszkedést a minta végétől kezdődően.

Az algoritmus működésének a szemléltetésére tekintsük a következő példát. A szöveg legyen: "BUDAPESTEN SOK A SZOBAFESTŐ."; a minta pedig a "FEST" szó.

Ekkor a mintát kezdetben a szöveg elejére helyezzük, azaz 'FEST' és 'BUDA' kerülnek fedésbe. A mintát jobbról balra ellenőrizve a T és A jelek kerülnek ellenőrzésre. Miután ezek különböznek ezért a mintát újra pozícionáljuk. Az A nem szerepel a mintában ezért azt átugorhatjuk ( skip[A] = M ), és az újra pozícionálás után az 'PEST' és a 'FEST' kerül fedésbe. Itt T és T kerül összehasonlításra, majd S és S, ezután E és E, végül P és F amelyek nem egyeznek, így mintát újra kell illeszteni. Ez most egy karakterrel történő eltolást jelent. Ekkor 'ESTE' és 'FEST' kerül fedésbe, és E-t és T-t kell összehasonlítani. Miután E szerepel a mintában ezért a mintát úgy kell elhelyezni, hogy abban az utolsó E fedje ezt a szövegbeli E-t, azaz kettővel kell jobbra tolni ( skip[E] = 2 ). Ezek után csupa ugrás következik egész addig amíg 'OBAF' és 'FEST' nem kerül fedésbe. Ekkor F és T összehasonlítása után a mintát 3 jellel el kell tolni ( skip[F] = 3 ), és ezután jobbról balra összevetve a jeleket a minta egy előfordulására találunk. Ezt a folyamatot szemlélteti a következő ábra.

```
B U D A|P E S T|E|N   |S O K   |A   S Z|O B A F|E S T|Ő .
      ≠|≠ = = =|≠|  ≠ |       ≠|      ≠|      ≠|
F E S T|F E S T|  |   |        |       |      =|= = =
        F E S T|  |   |        |       |       |
            F E S T|F E S T|F E S T|F E S T|
                                           F E S T
```

Az initskip program végzi el az előfeldogozást, azaz határozza meg a skip vektor értékeit minden H-beli elemre. Tegyük fel, hogy H-ra megengedettek a *first*, *succ*, *last* műveletek, amelyek H első elemét, egy H-beli elem rákövetkezőjét, illetve H utolsó elemét adják meg. Ezek alapján az initskip program:




Az algoritmus előnye, hogy nagy méretű ábécé esetén átlagosan hosszú részeket ugrik át, jó esetben N/M lépésben is a szöveg végére érhet. Minél nagyobb az ábécé annál valószínűbbek a hosszú ugrások, persze ez egyben a skip vektor méretét is növeli. Az algoritmus hátránya - az esetlegesen nagy skip vektoron kívül -, hogy szabad mozgást feltételez a szövegen, ezért szekvenciális fájlokra közvetlenül nem alkalmazható. (Megfelelően nagy pufferre van szükség ennek a feloldására.) Az is látható, hogy legrosszabb esetben az algoritmus N*M összehasonlítást is végrehajthat.

## Quick Search

Az algoritmus elveiben majdnem azonos a már bemutatott Boyer-Moore algoritmus alapelveivel, azaz itt is a szöveg lehetséges karaktereire nézve egy táblázatot készítünk, ami tartalmazza, hogy a mintát miként kell újra pozicionálni a szövegben, hogy illeszkedjen a megfelelő jelre.

Először a mintát a szöveg elejére illesztjük - általában az i. pozícióra - és ellenőrizzük az illeszkedését balról jobbra. Ha illeszkedik, akkor megtaláltuk az első előfordulást. Ha nem akkor a minta utáni első jelet (i+M.) kell ellenőrizni (miután a minta a jelenlegi M jelre biztosan nem illeszthető), és a legkisebb mértékben eltolni a mintát úgy hogy egy jele megegyezzen ezzel a jellel. Természetesen ennek csak akkor van értelme, ha a jel előfordul a mintában, különben a minta illesztését a szöveg következő jelétől (i+M+1.) kezdhetjük.

Vizsgáljuk meg az algoritmus működését a már ismert példán, azaz a szöveg legyen: "BUDAPESTEN SOK A SZOBAFESTŐ."; a minta pedig: "FEST". Ekkor először a B és F jeleket hasonlítjuk össze. Miután ezek nem egyeznek a minta utáni első jelet, P-t, vizsgáljuk. Ez nem szerepel a mintában, így átugorható, azaz a minta új helyzetében a következő E-nél kezdődik. Miután E és F különböző, ezért a minta utáni jel, N vizsgálandó. Ez nem fordul elő a mintában, így átugorható. Ez

folytatódik, amíg 'BOFE' és 'FEST' nem kerül fedésbe. Itt B és F különböző, azonban a minta utáni jel, S előfordul a mintában, ezért ezeket fedésbe kell hozni. Az összehasonlítások elvégzése után megtaláljuk a keresett előfordulást. Ezt a folyamatot szemlélteti a következő ábra.

```
|B U D A P|E S T E N|  S O K  |A   S Z O|B A|F E S T Ő.
|≠       ∉|≠       ∉|≠       ∉|≠       ∉|≠  |      ∈
|F E S T  |F E S T  |F E S T  |F E S T  |F E S T
                                            |F E S T
```

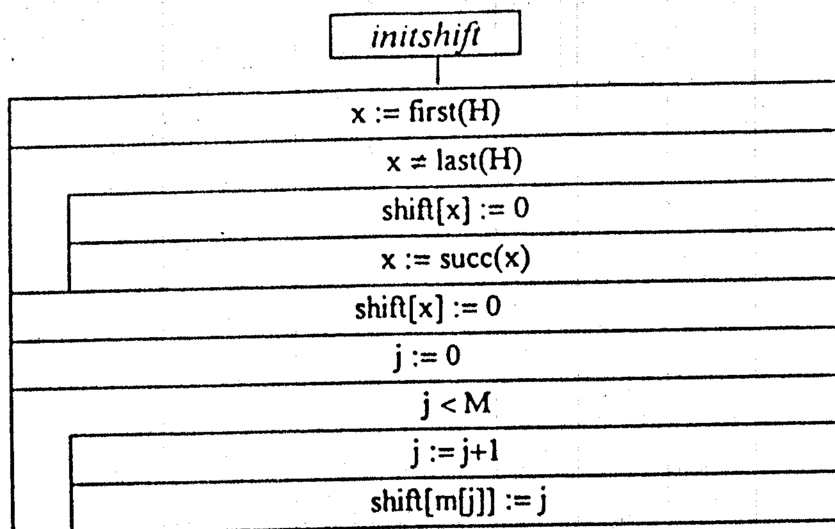
Definiáljuk a *shift* függvényt úgy, hogy egy karakterre határozza meg a minta utolsó pozícióját ahol az előfordul; illetve legyen 0 az értéke ha a jel nem szerepel a mintában. Formálisan:

$$\text{shift}: H \rightarrow [0..M], \text{ és}$$
$$\text{shift}(x) := 0 \text{, ha } x \notin m; \text{ illetve}$$
$$\text{shift}(x) := j, \text{ ahol } m[j] = x \text{ és } \forall k \in [j+1..M]: m[k] \neq x.$$

A shift függvényt az eddigiekhez hasonlóan egy vektorral helyettesítjük a programban. A vektor kiszámítását az initshift program végezze el. Ezek után a programban a szövegen haladunk végig (i), és megpróbáljuk a mintát egyre hosszabb szakaszokban (k) illeszteni az adott pozícióra. Ha a megfelelő jelek egyeznek, akkor növeljük a szakasz hosszát (k:=k+1), különben a minta új kezdőpozícióját határozzuk meg a shift segítségével a következőképpen. Jelenleg a minta az i. pozíción kezdődik, így s[i+M] az első utána következő jel. Erre a minta h = shift[s[i+M]] -dik jele illeszkedik, azaz s[i+M]=m[h] -t kell biztosítani, ami teljesül ha a minta első jele (m[1]) illeszkedik s[i+M+1−h] -ra. Ez azt jelenti, hogy i értékét M+1−shift[s[i+M]] -mel kell növelni és k értékét 0-ra állítani. Tehát a program, aminek a levezetését az olvasóra bízzuk:

## Quick Search

<table>
<tr><td colspan="3" align="center">initshift</td></tr>
<tr><td colspan="3" align="center">i,k := 1,0</td></tr>
<tr><td colspan="3" align="center">i ≤ N−M+1 ∧ k < M</td></tr>
<tr><td rowspan="2"></td><td colspan="2" align="center">\ s[i+k] = m[1+k] /</td></tr>
<tr><td align="center">k := k+1</td><td align="center">i,k := i+M+1−shift[s[i+M]],0</td></tr>
<tr><td colspan="3" align="center">u := (k = M)</td></tr>
</table>




A program egy kicsit javítható, ha shift helyett shift*-ot használunk benne, ahol shift*[x]=M+1−shift[x]. Ahogy az a bevezetőből kiderült az algoritmus tulajdonságai nagyon hasonlítanak a Boyer-Moore algoritmus jellemzőire. Egy kicsit hatékonyabb lehet a Quick Search hiszen nem a minta utolsó jelének megfelelő jel szerint ugrik a szövegben, hanem az azt követő szerint, ami nagyobb ugrásokat eredményezhet. (Nem feltétlen jelent nagyobb ugrást hiszen az ugrás mértéke a jeltől is függ!)

## Rabin-Karp algoritmus

Az algoritmus abból indul ki, hogy a string keresések nehézségét a stringek (a minta és egy szövegszelet) összehasonlításának a bonyolultsága okozza. Ezért áttérünk egy olyan állapottérre, ahol az egyenlőség eldöntése egyszerű, és azon a feladat megoldása egy egyszerű lineáris keresés. Az egyik legegyszerűbb feladat egész számok egyenlőségének az eldöntése, így logikus hogy az új állapottéren egy egész szám feleljen meg a mintának illetve egy szövegszeletnek.

Az állapottér transzformációt megadó függvény megadható, ha az M hosszúságú stringet egy olyan számrendszerben felírt számnak képzeljük el, amelynek a számjegyei az ábécé jelei. Ez azt jelenti hogy ha d jelünk van az ábécében, azaz d=|H|, akkor egy d alapú számrendszerben felírt számnak tekintjük a stringet. Ha rendelkezésünkre áll az *ord* függvény, ami megadja hogy egy jel hányadik az ábécében, akkor az i. pozíción kezdődő szövegszeletnek megfelelő szám értékét az alábbi formulával számolhatjuk ki:

$$g(s,i) = \sum_{j=i}^{i+M-1} ord(s[j]) * d^{i+M-1-j}$$

A minta értékét hasonlóan kapjuk meg s helyébe m-et és i helyébe 1-et helyettesítve. Nyilvánvaló, hogy a fenti g függvény egyértelmű, ezért a feladat alábbi specifikációja ekvivalens az eredetivel.

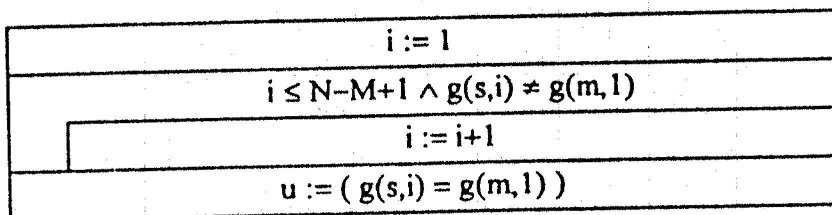
$$A = \underset{s}{\mathcal{S}} \times \underset{m}{\mathcal{M}} \times \underset{u}{\mathcal{L}} \times \underset{i}{\mathcal{N}}$$

$$B = \underset{s'}{\mathcal{S}} \times \underset{m'}{\mathcal{M}}$$

$$Q = (\, s = s' \wedge m = m' \,)$$

$$R = Q \wedge u = (\exists j \in [1..N-M+1]: g(s,j) = g(m,1)) \wedge$$
$$u \Rightarrow (i \in [1..N-M+1] \wedge g(s,i) = g(m,1) \wedge \forall j \in [1..i-1]: g(s,j) \neq g(m,1))$$

A megoldó program így nem más mint egy egyszerű lineáris keresés, azaz:




A fenti programban az összehasonlítások száma ugyan legfeljebb N-nel arányos, azonban ehhez szükséges a g függvény értékeinek ismerete. Az egyes értékek kiszámítása pedig M db sorzást igényel, ami még drágább művelet mint az összehasonlítás. Ezért első pillantásra a fenti megoldás nagyon rossznak látszik. Szerencsére a g függvény értékei könnyen számíthatók a szöveg soron következő szeletére, ugyanis az i. jelet ki kell venni mint legmagasabb értékűt, a maradékot balra léptetni (d-vel szorozni) és utolsó jelként az i+M. jelet hozzávenni. Tehát:

$$g(s,i+1) = (g(s,i) - ord(s[i]) * d^{M-1}) * d + ord(s[i+M])$$

Ez egy szorzást tartalmaz, így a leírt problémát kiküszöböli. Elméletileg a feladatot meg is oldhatnánk az eddigiek alapján, azonban a gyakorlatban ez nem mindig lenne megvalósítható, ugyanis hosszabb minták és nagy méretű ábécék esetén a g függvény által meghatározott szám nem lenne tárolható egészként (pl. longcardként Modulában). Ez pedig az egész állapottér transzformációt feleslegessé tenné. Ezen azonban könnyű segíteni ha az egyértelműséget nem akarjuk garantálni. Tegyük fel, hogy ismerünk egy kellően nagy prímszámot ami még 'kényelmesen' ábrázolható. Legyen ez $p$. Ennek segítségével definiáljuk $h$-t a következőképpen:

$$h(s,i) := g(s,i) \bmod p$$

Ez a h függvény ugyan nem egyértelmű, de majdnem az. Nevezetesen igen nagy valószínűséggel különböző értékeket ad különböző stringekre. Különösen igaz ez a gyakorlati alkalmazásokra. Az is látható, hogy h(s,i+1) értéke is könnyen számolható h(s,i) ismeretében, nevezetesen:

$$h(s,i+1) = [(h(s,i) - ord(s[i]) * d^{M-1}) * d + ord(s[i+M])] \bmod p$$

Ennek a formulának az egyetlen hibája, hogy $h(s,i) - ord(s[i]) * d^{M-1}$ lehet negatív, ami bizonyos esetekben megzavarhatja a mod függvényt. Ezért helyettesítsük a

kifejezésnek ezt a részét egy olyan kifejezéssel ami biztosan nem negatív és nem változtatja a maradékot. Ez megtehető ha a kifejezést d*p -vel növeljük, azaz:

$$h(s,i+1) = [(h(s,i)+d*p-ord(s[i])*d^{M-1})*d+ord(s[i+M])] \bmod p$$

Amikor p 'kényelmes' ábrázolhatóságáról beszéltünk tulajdonképpen arra kellett gondolnunk, hogy a fenti kifejezés még ábrázolható marad, azaz nem csordul túl a modulus képzés előtt. Szerencsére a modulus tulajdonságai lehetővé teszik, hogy kisebb egységekben számoljuk ki a fenti kifejezést, így nagyobb prím választható, mint egyébként. Egy célszerű és 'stabil' kiszámítási mód:

$$h = (h(s,i)+d*p-ord(s[i])*d^{M-1}) \bmod p$$
$$h(s,i+1) = h*d+ord(s[i+M])] \bmod p$$

Ezek után a feladat specifikációjának tekintsük a fenti specikációt g helyett h-val, és oldjuk meg ezt a feladatot. A program 3 rész szekvenciája lesz. Az elsőben kiszámoljuk a $d^{M-1}$ kifejezés értékét mod p (dh), hogy ezt ne kelljen minden egyes i-re megtenni. A második részben kiszámoljuk a h értékét (hm) a mintára, illetve a szöveg első (i=1) kezdőszeletére (hs). Ezután a harmadik rész tartalmazza a feladatot megoldó tulajdonképpeni lineáris keresést, amelyben hs értékét számoljuk újra a fentieknek megfelelően. (Tulajdonképpen felfogható egy speciális olvasó műveletnek - hs,s: read -, ami a következő számot olvassa a transzformált állapottérbeli számsorozatból. Ekkor a második rész hs-re vonatkozó része felel meg az előre olvasásnak.) A program levezetését az olvasóra bízzuk.

Rabin-Karp

```
dh,j := 1,1
j < M
    dh,j := d*dh mod p,j+1
hm,hs,j := 0,0,1
j ≤ M
    hm := (hm*d+ord(m[j])) mod p
    hs := (hs*d+ord(s[j])) mod p
    j := j+1
i := 1
i ≤ N-M+1 ∧ hm ≠ hs
    hs := (hs+d*p-ord(s[i])*dh) mod p
    hs := (hs*d+ord(s[i+M])) mod p
    i := i+1
```

Vegyük észre, hogy ez a program nem oldja meg az eredeti feladatot (minden esetben), ugyanis h nem egyértelmű. Ezért a megtalált szövegszeletet még ellenőrizni kell, hogy valóban illeszkedik-e a mintára. Ha nem, akkor a keresést tovább kell

folytatni. Szerencsére ez az eset rendkívül ritkán fordul elő, így azt lehet mondani hogy az algoritmus műveletigénye a gyakorlatban a szöveg hosszával arányos.

## Dömölky szűrő

Hasonlóan az előző megoldáshoz ebben az esetben is egy állapottér transzformációt hajtunk végre annak érdekében, hogy a stringek összehasonlítását elkerülve egy egyszerű lineáris kereséssel oldhassuk meg a feladatot. Most a legegyszerűbb állapottérre próbálunk áttérni, ahol logikai értékek egy sorozatán kell az első igaz értéket megtalálni. Azaz a sorozat i. értéke legyen igaz akkor és csak akkor ha a minta illeszkedik az eredeti szöveg i. jelénél, ahol i az illeszkedő szelet végére mutat.

A feladat ezek után nem más mint ennek a logikai értékekből álló sorozatnak az előállítása. Miután egy logikai érték és egy M hosszúságú minta információ tartalma között lényeges különbség van, nyilvánvaló hogy további adatokra lesz szükségünk annak érdekében, hogy az információ vesztést elkerülve kényelmesen meg tudjuk határozni az éppen esedékes logikai értéket. Ezért vezessük be az alábbi T típust, ami logikai értékekből álló vektor és a továbbiakban bitsorozatnak nevezzük.

$$T = \text{vekt}(\,[1..M],\ \mathcal{L}\,)$$

Tehát egy v *T* típusú változó elemei a bitek, azaz v[k] logikai érték a k. bit. Ezen a típuson legyenek értelmezve a szokásos vektor műveletek, és ezeken kívül vezessünk be két új típusműveletet a *jobbra*, illetve *és* műveleteket, amelyek a vektor bitjeit léptetik jobbra úgy, hogy a belépő bit igaz legyen, illetve két bitvektort 'éselnek' össze úgy, hogy az eredmény vektor egy bitje a megfelelő bitek közötti logikai és művelet eredménye. A műveletek formális specifikációja:

$$A_{jobbra} = \underset{x}{T} \times \underset{y}{T}$$

$$B_{jobbra} = \underset{x'}{T}$$

$$Q_{jobbra} = (\,x = x'\,)$$

$$R_{jobbra} = Q_{jobbra} \wedge \forall i \in [2..M]:\ y[i] = x[i-1] \wedge y[1] = \text{igaz}$$

$$A_{és} = \underset{x}{T} \times \underset{y}{T} \times \underset{z}{T}$$

$$B_{és} = \underset{x'}{S} \times \underset{y'}{M}$$

$$Q_{és} = (\,x = x' \wedge y = y'\,)$$

$$R_{és} = Q_{és} \wedge \forall i \in [1..M]:\ z[i] = x[i] \wedge y[i]$$

Vezessük be a *mask* függvényt ami minden ábécében szereplő jelhez egy bitsorozatot rendel úgy, hogy pontosan ott szerepeljenek igaz értékek ahol a mintában előfordul a jel. Azaz mask(x)[i] legyen igaz akkor és csak akkor ha m[i] = x. Ez a függvény tulajdonképpen az x jel bitmaszkja a mintára nézve. Formálisan:

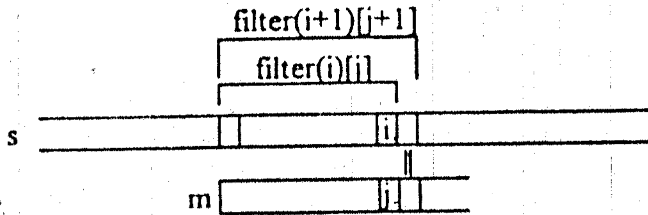
$mask: H \rightarrow T$, és
$mask(x) = t$, ahol $t[i] \Leftrightarrow (m[i] = x)$

Vezessük be továbbá a *filter* függvényt is, ami rendelkezzen azzal a tulajdonsággal, hogy filter(i) -nek a j. bitje pontosan akkor igaz, ha a minta a szöveg i. pozicióján j hosszon illeszkedik a szövegre úgy, hogy az i. az utolsó illeszkedő jel. Formálisan:

$filter: [0..N] \rightarrow T$,
$filter(i) = t$, ahol $t[i] \Leftrightarrow match^{*}(i,j)$

Nyilvánvaló, hogy ezek után a feladat ekvivalens a filter(i)[M] logikai értékek sorozatában az első igaz érték megtalálásával, azaz megtaláltuk a szükséges állapottér transzformációs függvényt. Már csak ennek a függvénynek az értékeit kell kényelmesen kiszámolnunk.

Vizsgáljuk meg, hogy filter(i) j. bitjének ismeretében miként számolhatjuk ki filter(i+1) j+1. bitjét. Ez utóbbi bit pontosan akkor igaz, ha a minta az i. pozíción j hosszon illeszkedett - azaz filter(i)[j] -, és a szöveg i+1. jele megegyezik a minta j+1. jelével - azaz s[i+1] = m[j+1] -.




Az s[i+1] = m[j+1] feltétel ekvivalens azzal, hogy az s[i+1] jelhez tartozó bitmaszk j+1. bitje igaz-e, azaz mask(s[i+1])[j+1] értékével. Ez azt jelenti, hogy filter(i+1) j+1. bitje megkapható úgy, hogy filter(i) j. bitjét eggyel jobbra léptetjük a sorozatban és 'hozzáéseljük' mask(s[i+1])-et. Ez j > 1 esetén minden bitre igaz hiszen semmi speciálisat sem használtunk ki j-ről; j = 1 esetére pedig könnyen ellenőrizhető, hogy pontosan akkor illeszkedik a minta egy hosszon ha m[1] = s[i+1], azaz mask(s[i+1])[1], és pontosan ezt eredményezi a fenti művelet. Tehát:

$$filter(i+1) = és(\, jobbra(filter(i)), mask(s[i+1])\,), \quad i \in [0..N-1]$$

Tehát filter egy rekurzív függvény, és csak filter(0)-t kell megadni, hogy teljesen definiálva legyen. Könnyen ellenőrizhető, hogy filter(0) := <u>false</u> megfelelő, hiszen a nulladik pozición a minta semmilyen hosszon sem illeszkedhet a szövegre. (<u>false</u> jelenti a csupa hamis értékből álló vektort.)

A transzformált állapottéren a filter függvény felhasználásával a feladat a következőképpen specifikálható:

$$A = S \times M \times L \times N$$
$$\quad s \quad m \quad u \quad i$$

$$B = S \times M$$
$$\quad s' \quad m'$$

$$Q = ( s = s' \land m = m' )$$

$$R = Q \land u = (\exists j \in [1..N]: filter(j)[M]) \land$$
$$u \Rightarrow (i \in [1..N] \land filter(i)[M] \land \forall j \in [1..i-1]: \neg filter(j)[M])$$

A megoldó program visszavezethető lineáris keresésre, ahol a feltétel filter(i)[M]. A programban a filter függvényt helyettesítjük az *f* változóval, és a rekurzív függvény kiszámítását invertáljuk a lineáris kereséssel. Továbbá a szokásos módon a mask függvényt helyettesítsük egy vektorral, aminek kiszámítását végezze az *initmask* program. (Ennek megírása az olvasó feladata!) Ezek után a megoldó program:

Dömölky-szűrő

| *initmask* | |
|---|---|
| f,i := false,0 | |
| i < N ∧ ¬f[M] | |
|     f,i := és( jobbra(f), mask(s[i+1]) ),i+1 | |
| \ f[M] / | |
| i,u := i−M+1,true | u := false |

Látható, hogy a program a szöveg minden jelét egyszer vizsgálja meg, így soha sem igényel N-nél több összehasonlítást. Ezen kívül a jeleken szekvenciálisan halad végig, tehát gond nélkül adaptálható szekvenciális fájlokra. Ha a minta nem túl hosszú (nem több, mint 32 jel), akkor a bitsorozat *és* illetve *jobbra* műveletei könnyen és hatékonyan implementálhatók, hiszen ezek megfelelői léteznek egy gépi szóra. Az algoritmus gyenge pontja a mask vektor, ugyanis ennek inicializálása nagy méretű ábécék esetén viszonylag költséges lehet. Az algoritmust akkor célszerű alkalmazni, amikor a mask inicializálásának műveletigénye elhanyagolható a kereséshez képest. Tehát kisebb ábécék esetén, illetve ha több előfordulás megtalálása a cél. Nem okoz nehézséget úgynevezett 'wild card'-os keresések megvalósítása sem, hiszen ekkor a megfelelő bitet az összes maszkban igazra kell állítani. (Wild card azt jelenti, hogy egy pozíción bármilyen jel állhat.)