Adattömörítés

Bevezetés

Az adattömörítés feladata egy adott információ minél kisebb helyigényű leírása. Erre nyilvánvalóan szükség van ha egy adott tároló egységet gazdaságosan szeretnénk kihasználni, vagy nagy mennyiségű adatot szeretnénk lehetőleg olcsón és gyorsan továbbítani. (Gondoljunk nagy méretű fájlok hálózati elküldésére vagy beszerzésére, illetve adatok floppyra másolására.) Fontos követelmény a tömörítések esetén az, hogy a tömörített leírásból visszanyerhessük az eredeti adathalmazt. Bizonyos esetekben követelmény, hogy a visszaalakítás során információ ne vesszen el, és vannak olyan esetek is, amikor valamilyen mértékű információ vesztés megengedett. (Ez utóbbiakat adatvesztéses tömörítéseknek nevezzük. Ilyen jellegű tömörítéseket használnak például a képfeldolgozó rendszerekben - digitális képtovábbítás -, ahol az adatvesztés még lehetővé teszi jó minőségű képek előállítását.) A továbbiakban csak olyan tömörítésekkel foglalkozunk, ahol nem lép fel információ veszteség a tömörítés, illetve a visszaalakítás során. A tömörítési szakaszt *kódolás*nak, a visszaalakítást pedig *dekódolás*nak nevezzük.

Legyen H egy véges halmaz, az ábécé, elemszáma d; $B := \{0,1\}$, a bitek halmaza. Ekkor a tömörítés feladata egy H-beli sorozat lekódolása minimális számú bittel, azaz minimális hosszúságú B-beli sorozattal úgy, hogy az dekódolható legyen. Tehát adattömörítésnek nevezünk egy f: $H^* \to B^*$ invertálható függvényt.

Számítógépen az ábécé elemeit is bitsorozatokkal ábrázoljuk, így valójában $B^* \to B^*$ típusú függvényekről van szó. Azt szeretnénk a tömörítés során elérni, hogy a kódolt bitsorozat rövidebb legyen mint az eredeti, lehetőleg minden esetben.

Könnyen látható, hogy a fenti feltételek mellett nem lehet olyan függvényt találni, ami minden sorozathoz rövidebb sorozatot rendel. Ez ugyanis azt jelentené, hogy a függvény ismételt alkalmazásával minden sorozat végső soron leképezhető lenne egy egy hosszúságú sorozatra (sőt üres sorozatra), ami nyilvánvaló információ veszteséget jelentene. Ennél több is igaz, nevezetesen a következő tétel:

Tétel: Ha egy invertálható függvény minden bitsorozathoz legfeljebb önmagával megegyező hosszúságú bitsorozatot rendel, akkor minden sorozathoz pontosan önmagával megegyező hosszúságú sorozatot rendel; azaz nem tömörít.

Bizonyítás: Legyen B^n az összes legfeljebb n hosszúságú bitsorozatból álló halmaz. Ekkor f: $B^n \to B^n$ típusú függvény, hiszen f minden sorozathoz legfeljebb önmagával megegyező hosszúságú sorozatot rendel, azaz semmilyen B^n-beli sorozathoz sem rendelhet n-nél hosszabb sorozatot. Ugyanakkor f invertálható és az alaphalmaz és a képhalmaz megegyezik ($D_f=R_f=B^n$), ami csak úgy lehet, hogy a képhalmaz ($R_f=B^n$) minden értéke felvétetik értékként. Ekkor viszont ha f egy B^n-beli sorozathoz rövidebb sorozatot rendelne, akkor lennie kellene olyan sorozatnak is amihez hosszabbat rendel, hiszen az adott hosszúságú sorozatok száma rögzített.

A fenti tételből az következik, hogy nem létezik olyan tömörítő algoritmus ami tetszőleges sorozatot tömörít információ vesztés nélkül, akármilyen kis mértékben is. A gyakorlatban azonban mégis léteznek tömörítő programok. Felmerül a kérdés, hogy

mi ad lehetőséget a tömörítésre? A gyakorlati esetek fontos jellemzője, hogy nem minden n hosszúságú sorozatot kell tömöríteni, hanem azoknak csak egy (rendszerint kicsi) részhalmazát. Erre a részhalmazra még továbbá az is igaz, hogy a benne előforduló sorozatokban vannak szabályosságok, ismétlődések, illetve a jelek különböző gyakorisággal szerepelnek. Ezek a tulajdonságok teszik lehetővé, hogy a gyakorlatban előforduló sorozatokat tömöríteni lehet.

Egy egyszerű tömörítési eljárás az amikor az ismétlődő jeleket nem írjuk ki többszörösen, hanem megadjuk, hogy hányszor fordult elő és utána a jelet. Tehát ha a kódolandó sorozat az 'AAAAABBCCCDDDD...', akkor ezt a '5ABB3C4D...' sorozattá alakítjuk. Most feltettük, hogy az eredeti sorozatban nem szerepelnek számjegyek, ugyanis csak ebben az esetben alakítható vissza egyértelműen a kódolt szöveg. (BB-t nem érdemes 2B-vé alakítani, ugyanis ez nem jelent rövidítést.) Ha ez nem igaz, akkor speciális jel (pl. #) bevezetése szükséges, ami jelzi, hogy ismétlődési tényezőt reprezentál a következő jel. Az algoritmus további problémáinak bemutatásától, illetve a lehetséges megoldások megadásától eltekintünk, azt az olvasóra bízzuk.

Egy másik lehetséges tömörítési eljárást eredményez az az észrevétel, hogy a szöveg jeleit redundánsan ábrázoljuk a számítógépen. Azaz egy karakteres szöveg minden egyes jelét 8 biten ábrázoljuk. Ez nyilván felesleges, ha a szövegben csak az angol ábécé nagybetűi és a szóköz fordulnak elő. Ez összesen 27 jel, amelyeket meg tudunk különböztetni 5 bit felhasználásával. Így egy ilyen típusú n hosszúságú szöveg kódolható egy 5*n hosszúságú bitsorozattal az eredeti 8*n helyett.

Sajnos a gyakorlatban ilyen jelentős mértékben nem szűkül le a jelkészlet, így a fenti módszer nem vezet jó eredményre. A módszer alapötlete, miszerint próbáljuk rövidebb bitsorozatokkal ábrázolni az egyes jeleket, azonban továbbfejleszthető. Ne törekedjünk arra, hogy minden jelet rövidebben és ugyanannyi bittel reprezentáljunk, ehelyett használjunk változó hosszúságú bitsorozatokat. Rendeljünk rövid sorozatokat a sokszor előforduló jelekhez és hosszabb sorozatokat a ritka jelekhez. Így remélhetőleg a teljes bitsorozat hossza is rövidebb lesz, hiszen a gyakori jelek kevés helyet foglalnak és ez a megtakarítás várhatóan felülmúlja a ritka jeleknél fellépő veszteséget.

Legyen a kódolandó szöveg: 'FELESLEGES'. Ez 10 betűből áll, ami karakteres ábrázolás esetén 80 bitet jelent. (Kihasználva, hogy csak 5 jel szerepel benne, amiket 3 bit segítségével megkülönböztethetünk a sorozat kódolható 10*3 = 30 bittel. Ekkor azonban a dekódoláshoz ismerni kell a leíró táblázatot is, ami megadja, hogy melyik bithármas melyik jelnek felel meg.)

Ha megszámoljuk, hogy az egyes jelek hányszor fordulnak elő a szövegben, akkor azt kapjuk, hogy az E négyszer, a L kétszer, az S kétszer, a F és a G pedig egyszer-egyszer szerepel. Próbáljuk meg tehát minél rövidebb bitsorozatokat rendelni ezekhez a jelekhez a fenti gyakoriságok szerint. Az E szerepel a legtöbbször, így rendeljük hozzá a lehető lerövidebb, egy jelből álló '0' bitsorozatot. Folytassuk a hozzárendelést és ábrázoljuk a második leggyakoribb jelet, L-t, az '1'-gyel, és így tovább az 'S'-et '01'-gyel, a 'F'-t '10'-val és a 'G'-t '11'-gyel. Azaz a következő táblázat adja meg a kódolást:

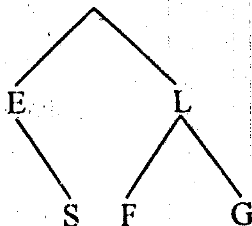
E	0
L	1
S	01
F	10
G	11

Ekkor a sorozat kódja: 10 0 1 0 01 1 0 11 0 01. Szóközökkel választottuk el az egyes jeleknek megfelelő biteket. Ebben az esetben az elhatároló jelek használatával a sorozat egyértelműen dekódolható a táblázat ismeretében. Elhatároló jelek nélkül az '10010011011001' sorozatot kapjuk, ami 14 bit. Azonban ebből már nem tudjuk megadni az eredeti sorozatot, hiszen a 'FELESLEGES' mellett egy lehetséges értelmezés a 'FSEEGELFS', és még számos más sorozat is dekódolható a fenti táblázatot használva. Ezért ez a kódolás nyilván elfogadhatatlan.

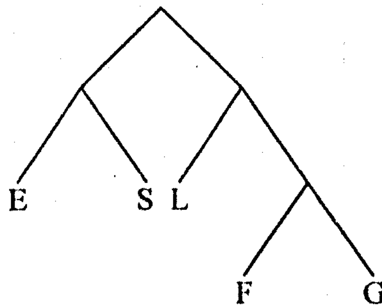
Egy megoldási módszer lehet szeparátor jelek bevezetése, ami az egyes jeleknek megfelelő bitsorozatokat elválasztja. Ekkor a dekódolás egyértelmű lesz. Ezt tettük az előzőekben a szóközök alkalmazásával. Ez azonban jelentősen növeli a kódolt szöveg hosszát. Egy jobb módszerhez jutunk akkor, ha megvizsgáljuk, hogy miért értelmezhető a kódolt sorozat többféleképpen, és ezt próbáljuk kijavítani.

A fenti esetben a többértelműség oka, hogy egy bit olvasásakor nem tudjuk eldönteni, hogy az egy jel utolsó bitje-e vagy pedig egy másik jel kezdőszeletéhez tartozik. Azaz az okozza a problémát, hogy egy kódszó egy másik kódszó kezdőszelete, prefixe. (Pl. a 0 az E kódja, de az S kódjának első bitje is egyben.) Ha sikerülne olyan kódszavakat rendelni az egyes jelekhez, amelyek közül egyik sem prefixe semelyik másiknak sem, akkor a kódolt sorozat egyértelműen dekódolható lenne. Ilyen kódolás esetén ugyanis egy jel olvasásakor el tudjuk dönteni, hogy az egy kódszó vége-e vagy sem, a kódszavak pedig egyértelműek.

Ábrázoljuk a fenti kódtáblázatot egy bináris fával. (Ez bináris kódok, azaz bitsorozatok esetén mindig megtehető.) 0 feleljen meg egy bal oldali ágnak, 1 pedig egy jobb oldali ágnak a fában. Írjuk a fa csúcspontjaiba a jeleket, mégpedig úgy, hogy a gyökérből a csúcsba vezető út feleljen meg a jelhez rendelt kódszónak. (Így nyilván minden csúcsnak legfeljebb egy címkéje lesz.) Ennek alapján a fenti táblázatnak megfelelő fa:



Látható, hogy ebben a fában a kódszavak prefixére kimondott feltétel ekvivalens azzal, hogy egy jel sem helyezkedig el egy másik jelhez vezető úton. (Itt most az E az S-hez vezető úthoz tartozó csúcs címkéje, azaz az E kódja prefixe az S kódjának.) Ez úgy és csak úgy teljesíthető, ha csak a fa leveleihez rendelünk jeleket. Egy ilyen lehetséges fát mutat be az alábbi ábra (a fenti átalakítását), a hozzá tartozó kódtáblázattal együtt:

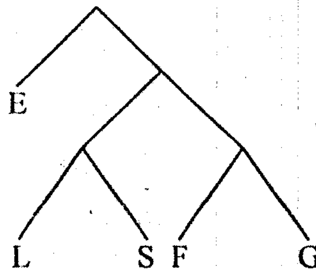


E	00
S	01
L	10
F	110
G	111

Ennek megfelelően kódolva a szöveget az '1100010000110001110001' 22 bit hosszúságú sorozatot kapjuk, amit egyértelműen lehet visszakódolni a táblázat, illetve a kódfa ismeretében. Ez azon múlik, hogy egy bináris fában minden levélhez egy és csak egy út vezet a gyökérből. Ez adja a dekódolás algoritmusát is, nevezetesen a gyökérből indulva balra illetve jobbra lépünk aszerint, hogy az olvasott jel 0 vagy 1, és ha levélhez érünk, akkor az annak megfelelő jelet kiírjuk, és újra indulunk a gyökérből.

Hátramaradt még annak a kérdésnek a tisztázása, hogy a lehetséges bináris fák közül melyiket válasszuk ki a kódoláshoz úgy, hogy a kódolt üzenet a lehető legrövidebb legyen. Esetünkben az alábbi kódfa szintén egy 22 bites kódot ('1100100010110001110101') eredményez, azonban az eddigiek alapján nem tudhatjuk, hogy lehet-e ennél jobb kódfát találni. Az optimális kódfa megtalálására szolgál a Huffman algoritmus.

nem igaz!! de igen!

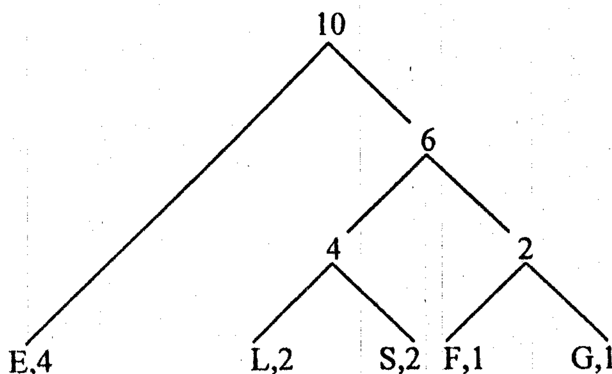


Huffman kód

Az optimális kódfa előállításához a Huffman algoritmus a jelek gyakoriságát használja fel, azaz ismertnek tekintjük a jelek előfordulásainak a számát a szövegben. Vegyünk fel minden jelet a relatív gyakoriságával együtt egy gráf csúcspontjaként. Ezek a csúcsok lesznek a létrehozandó kódfa levelei. Vonjuk össze a két legkisebb gyakoriságú csúcspontot egy közös csúcspontba. A közös csúcspont legyen őse mindkét összevont csúcsnak; az egyik legyen a bal oldali utódja, a másik pedig a jobb oldali. Az új csúcs gyakorisága legyen az eredeti gyakoriságok összege. Az összevont csúcsokat hagyjuk ki a további összevonásokból. Folytassuk ezt az eljárást addig, amíg egyetlen csúcsunk marad. Ez a csúcs lesz a kódfa gyökere.

Nyilvánvaló, hogy a fenti algoritmus eredménye egy bináris fa, aminek a levelei a jeleknek felelnek meg. Belátható, hogy ez a fa egy optimális kódfához vezet; továbbá, hogy a kód hossza független attól, hogy az egyes lépésekben az azonos gyakoriságú osztályok közül melyeket választunk ki összevonásra.

A már bemutatott példánál maradva a megadott optimális kódfa az alábbi módon áll elő:



Itt először az 'F'-nek és 'G'-nek megfelelő csúcsokat vontuk össze, hiszen ezek gyakorisága a legkisebb (1), és létrehoztunk egy őscsúcsot 2 gyakorisággal. Ezek után egy 4 és három 2 gyakoriságú csúcsunk van. Most az 'L'-nek és 'S'-nek megfelelő csúcsokat vontuk össze, és így keletkezett az ősük, aminek a gyakorisága 4. Ekkor három csúcs közül lehet választani, amelyek gyakorisága 4, 4 illetve 2. Az egyik csúcsnak a 2 gyakoriságú csúcsot kell választani a két négyes közül az összevontat használtuk fel ebben az esetben; az eredmény csúcs gyakorisága 6. A végső lépésben ezt kell összevonni az 'E'-nek megfelelő 4 gyakoriságú csúccsal, így kapjuk a fenti fát. (Nyilvánvalóan más fát kapunk, ha a második vagy a harmadik összevonási lépésben különböző csúcsokat választunk; azonban ellenőrizhető, hogy minden lehetséges esetben a végső kódsorozat hossza 22 bit.)

A kódfát létrehozó programban szükségünk lesz egy prioritás sorra, amibe elemek helyezhetők el megadott prioritással (*insert*), és a legalacsonyabb prioritású elem kivehető a sorból (*remove*). (A sor elemei lesznek a választható csúcspontok a megfelelő gyakoritás értékkel, mint prioritás.) Két további műveletet kell értelmeznünk: egy üres sor létrehozását (*init*), és a sor ürességének eldöntését (*empty*). Ennek alapján a prioritás sor típusspecifikációja:

$\boldsymbol{PQ} = (\boldsymbol{P}, \boldsymbol{I}, \{\text{init}, \text{empty}, \text{insert}, \text{remove}\})$, ahol:

$\boldsymbol{P} = \text{set}(\boldsymbol{N}, \boldsymbol{N})$ ($\boldsymbol{N}$ a természetes számok halmaza),

$\boldsymbol{I} = \text{true}$.

$\mathbf{A}_{\text{init}} = \boldsymbol{P}$ $\mathbf{B}_{\text{init}} = \{\alpha\}$

$\quad \text{p}$

$\mathbf{Q}_{\text{init}} = \text{igaz}$

$\mathbf{R}_{\text{init}} = (\text{p} = \varnothing)$

$A_{empty} = \underset{p}{P} \times \underset{l}{L}$

$B_{empty} = \underset{p'}{P}$

$Q_{empty} = p = p'$

$R_{empty} = Q_{empty} \wedge l = (p = \emptyset)$

$A_{insert} = \underset{p}{P} \times \underset{pr}{N} \times \underset{el}{N}$

$B_{insert} = \underset{p'}{P} \times \underset{pr'}{N} \times \underset{el'}{N}$

$Q_{insert} = p = p' \wedge pr = pr' \wedge el = el'$

$R_{insert} = (pr = pr' \wedge el = el' \wedge p = p' \cup (pr,el))$

$A_{remove} = \underset{p}{P} \times \underset{el}{N}$

$B_{remove} = \underset{p'}{P}$

$Q_{remove} = p = p' \wedge p \neq \emptyset$

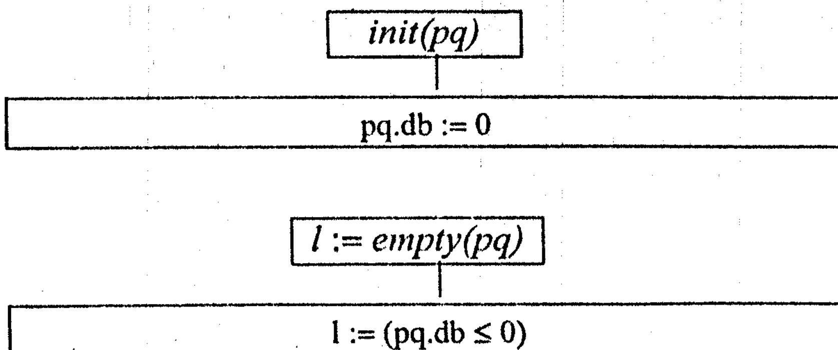
$R_{remove} = \exists (pr,e) \in p' \colon \forall (r,f) \in p' \colon pr \leq r \wedge e = el \wedge p = p' \setminus (pr,el)).$

Ennek a típusnak egy lehetséges megvalósítása az, amikor azt egy rendezett hármassal reprezentáljuk, aminek az első két komponense egy-egy vektor, a harmadik komponens pedig a sorban levő elemek száma (db). Az első vektor (e) a sorban levő elemeket tartalmazza, a második (p) pedig a megfelelő prioritásokat. Azaz:

$PQ = (e{:}E,\ p{:}P,\ db{:}N_0)$; ahol:

$E = \text{vekt}([1..d],N)$; $P = (\text{vekt}[1..d],N)$;

A típus részletes kifejtésétől eltekintünk, azt az olvasóra bízzuk. Az egyes műveleteket megvalósító programok:



```
insert(pq,pr,el)
  pq.db := pq.db+1
  pq.e[pq.db] := el
  pq.p[pq.db] := pr
```

```
el := remove(pq)
  min,j := 1,2
  j ≤ pq.db
    \   pq.p[j] < pq.p[min]   /
      min := j      |   SKIP
    j := j+1
  pq.e[min],pq.e[pq.db] := pq.e[pq.db],pq.e[min]
  pq.p[min],pq.p[pq.db] := pq.p[pq.db],pq.p[min]
  el := pq.e[pq.db]
  pq.db := pq.db–1
```

Tegyük fel, hogy a H ábécén értelmezett az *ord* függvény, ami egy jel sorszámát adja meg, azaz ord: H → [1..d]. A bináris fát két vektor segítségével reprezentáljuk. Az *ős* vektor megadja egy adott indexű csúcs szülőjének indexét, ami az index ha a csúcs bal oldali utód, illetve az index –1-szerese, ha a csúcs jobb oldali utód. A *gyak* vektor a csúcsoknak megfelelő gyakoriság értékeket tartalmazza. A fában levő csúcsok száma nem haladhatja meg a jelek számának a kétszeresét, azaz 2∗d-t. Tehát:

ős = vekt([1..2∗d],$\mathbb{Z}$); és gyak = (vekt[1..2∗d],$\mathbb{N}_0$).

A kódfa építés során kezdetben mindkét vektor első d eleme a jeleknek megfelelő csúcsokat ábrázolja. Ehhez kerülnek hozzá egyesével az összevont csúcsok az ős illetve gyak vektor soron következő elemét állítva, egészen addig amíg a prioritás sor ki nem ürül. Az új elemek bekerülnek a prioritás sorba, kivéve ha az új elem lenne az egyetlen elem.

Az alábbiakban megadjuk a kódfa építés programját, ahol feltesszük, hogy a gyak vektor első d eleme megfelelően inicializált, valamint, hogy a szövegben legalább két különböző jel szerepel. (A program megfelel a fenti leírásnak, az első része a prioritás sor inicializálása.)

Kódfa építés

```
init(pq); i := 1;
i ≤ d
    \ gyak[i] > 0 /
    insert(pq,gyak[i],i) | SKIP
    i := i+1
l := empty(pq)
¬l
    t1 := remove(pq)
    t2 := remove(pq)
    ős[i] := 0; ős[t1] := i; ős[t2] := –i;
    gyak[i] := gyak[t1]+gyak[t2]
    l := empty(pq)
    \ l /
    SKIP | insert(pq,gyak[i],i)
    i := i+1
```

A kódfa alapján a kódtáblázat elkészíthető. Ábrázoljuk az egyes jeleknek megfelelő kódokat két természetes számmal. Az első szám legyen a kód kettes számrendszerbeli értéke, a második pedig, hogy ebből hány bit számít, azaz a kód hossza. (A vezető nulláknak is fontos szerepük van most.) Minden egyes jelre az első számot, a kód értékét, tároljluk a *kód* tömbben, a hosszát pedig a *hossz* tömbben. Azaz:

$$kód = vekt([1..d], \mathcal{N}_0); \qquad \text{és} \qquad hossz = (vekt[1..d], \mathcal{N}_0).$$

A kódtáblázat előállítása nem más, mint a kódfában a levéltől a gyökérig vezető út megkeresése az ős vektor alapján minden jelre. Az út keresése során a megfelelő kettes számrendszerbeli számot kell előállítani, azaz ahogy haladunk felfelé úgy, kell a kettőhatványt növelni és ezt hozzáadni az eddigi értékhez ha jobb oldali utód volt vagyis a bit értéke 1 (ős érték negatív), illetve figyelmen kívül hagyni ha bal oldali utód azaz a bit értéke 0. A kód hossza a gyökérbe vezető út hossza lesz. Ezt az útkeresést végezze a *kódgen* program, amiben h jelölje az aktuális út hosszát, x a kód értékét és b a bit értékét. Ezt felhaszálva a kódtáblázat építő program:

Kódtábla

```
i := 1
i ≤ d
    h,x := 0,0
    \ gyak[i] > 0 /
    kódgen | SKIP
    kód[i],hossz[i] := x,h
    i := i+1
```

kódgen

```
t,b := ős[i],1
t ≠ 0
    \ t < 0 /
    x,t := x+b,–t | SKIP
    t,b,h := ős[t],2*b,h+1
```

A kódtábla segítségével a szöveg kódolható, nevezetesen a jeleket kell olvasni egyenként, a jelnek megfelelő bitsorozatot kiolvasni a táblából és ezt bitenként kiírni. A kódolt sorozathoz a kódtáblát ki kell írni (hossz és kód vektorok), enélkül az nem dekódolható. (Ez a kódolt sorozatot megelőzi.)

A dekódolás során a kódtábla alapján először fel kell építeni a kódfát. Ezután biteket kell olvasni és a gyökérből kiindulva a kódfa megfelelő ágain kell lefelé haladni, amíg levélhez nem érünk. Ekkor a levélnek megfelelő jelet ki kell írni, és újra a gyökérből indulni. A kódolás és dekódolás programját nem adjuk meg, annak megírása az olvasó feladata. (A bináris fa reprezentációjához az Adatszerkezetek c. tantárgy ad útmutatást.)

Aritmetikai tömörítés

A Huffman kód esetében a tömörítendő szöveg minden egyes jeléhez egy bináris kódszót rendeltünk, és ezek segítségével kódoltuk a szöveget. Próbálkozzunk egy ennél mohóbb módszerrel, és kíséreljünk meg az egész szöveghez egyetlen kódszót rendelni. Legyen ez a kódszó egy a [0,1] intervallumba eső racionális szám, amelyet az ábécé jeleinek relatív gyakorisága alapján számítunk ki a szövegre.

Legyen P a relatív gyakoriságokat megadó függvény, azaz $P: H \rightarrow (0,1)$. (P értékeit kiszámolhatjuk ha ismerjük a jelek gyakoriságát és a szöveg hosszát, hiszen egy adott jelre P értéke ezek hányadosa.) A relatív gyakoriságok alapján elkészítjük a $[0,1)$ intervallum egy teljes és diszkrét intervallum felbontását, méghozzá úgy, hogy

minden ábécébeli jelhez pontosan egy intervallumot rendelünk. Mindegyik részintervallum balról zárt, jobbról nyílt legyen.

Az első részintervallum bal oldali végpontja legyen 0, jobb oldali végpontja pedig az első jel relatív gyakorisága. Ez feleljen meg az első jelnek. Minden további intervallum bal oldali végpontja legyen az előző intervallum jobb oldali végpontja, a jobb oldali végpont pedig a bal oldali végpont növelve az ábécé következő jelének relatív gyakoriságával. Ez azt jelenti, hogy az i. intervallum bal oldali végpontja jelek relatív gyakoriságainak összege i–1-ig, a jobb oldali végpont pedig ugyanezen összeg i-ig. Ez az intervallum feleljen meg az ábécé következő, i. jelének. Formálisan, ha $\{I_1,...,I_d\}$ a részintervallumok halmaza, akkor:

$$I_i := [\ \sum_{j=1}^{i-1} P(char(j))\ ,\ \sum_{j=1}^{i} P(char(j))\ ); \qquad (i = 1,...d)$$

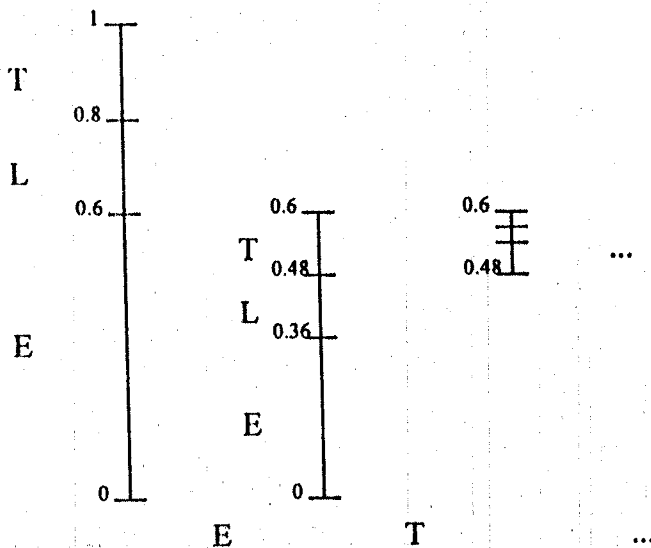
Legyen a tömörítendő szöveg az 'ETELE'. Ekkor az ábécé jeleinek gyakorisága: E esetén 3, L és T esetén pedig 1. A szöveg hossza 5, így a relatív gyakoriságok: P(E)=0.6, P(L)=0.2, P(T)=0.2. A részintervallumok: [0,0.6) felel meg E-nek, [0.6,0.8) felel meg L-nek, és [0.8,1) felel meg T-nek.

A szöveg első jelét egyértelműen kódolhatjuk a neki megfelelő intervallummal, pontosabban egy, az intervallumba eső tetszőleges racionális számmal. Továbbá az is igaz, hogy ha a teljes szöveghez olyan kódszót rendelünk, ami ebbe az intervallumba esik, akkor a dekódoláskor az első jel egyértelműen meghatározott. A szöveg második jeléhez rendeljünk egy intervallumot, ami azonban már nem a [0,1) része, hanem az első jel által meghatározott intervallum tovább finomítása. Azaz készítsük el az első jelnek megfelelő részintervallum egy teljes és diszjunkt felosztását a relatív gyakoriságok alapján, hasonlóan mint azt a [0,1) esetén tettük. Válasszuk ki ezen intervallumok közül a második jelnek megfelelőt, és tekintsük ezt az intervallumot a szöveghez rendelt kódnak. Ezzel az első jel még mindig visszafejthető, hiszen nem hagytuk el azt az intervallumot, és a második jel is megkapható, hiszen a dekódoláskor csak az első jel intervallumát kell felosztani a relatív gyakoriságok alapján, és kikeresni, hogy melyikbe esik a kód.

Ezt az eljárást lehet folytatni a szöveg jelein sorban végig haladva, egyre finomodó intervallum felosztások meghatározásával. A végső intervallumba eső tetszőleges racionális számmal kódolhatjuk a szöveget. A dekódolás során az egyes jelek sorban előállíthatók az intervallum felosztás újra számításával. Azonban ekkor nem tudjuk megmondani, hogy az eredeti szöveg mikor ér véget, hiszen az intervallumok finomítása tetszőlegesen sokáig elvégezhető. Ezért egy speciális terminális jel bevezetése vagy a szöveg hosszának megadása szükséges. (Speciális jel esetén amikor először a neki a megfelelő intervallumba érünk leállunk a dekódolással, a hossz ismeretében pedig megfelelő számú jel előállítása után kell abbahagyni a dekódolást.)

Nézzük meg, hogy mit jelent ez a már bemutatott példa esetén. (Lásd ábra.) Az első jel az E, az ennek megfelelő intervallum a [0,0.6). Ezt kell újra felosztanunk a relatív gyakoriságok alapján. Így kapjuk a [0,0.36), [0.36,0.48), [0.48,0.6) intervallumokat amelyek rendre megfelelnek az E, L és T jeleknek. A második jel a T, így az ennek megfelelő intervallumot, a [0.48,0.6)-ot kell újra felosztani. A felosztás eredménye a [0.48,0.552), [0.552,0.576), [0.576,0.6) intervallumok, amelyek az E, L és T jeleket reprezentálják. Ezt kell tovább folytatni, és végül a szövegnek megfelelő intervallum a

[0.5232,0.53184) lesz. Reprezentáljuk ezt az intervallumot a bal oldali végpontjával, azaz a szöveg kódja 0.5232 lesz.



A dekódoláskor a 0.5232 a [0,0.6)-ba esik, így az első jel csak az E lehet. Ezt az intervallumot újra felosztva látjuk, hogy a kód a [0.48,0.6)-ba esik, így a második jel csak a T lehet. Ezt az eljárást tovább folytatva eljutunk az 'ETELE' szövegig. (Ha nem tudnánk a szöveg hosszát, akkor termináló jel hiányában tovább lehetne folytatni a dekódolást és hamis jeleket kapnánk a szöveg végén.)

Vezessük be az *r* vektort az intervallumok jobb oldali végpontjainak a tárolására, azaz a kummulált relatív gyakoriságok tárolására. Egészítsük ki ezt a vektort egy nulladik elemmel, aminek az értéke legyen 0. Tehát:

$$\boldsymbol{R} = \text{vekt}([0..d], \boldsymbol{Q})$$

$$I_R(r) = (r[0] = 0 \wedge \forall i \in [1..d]: r[i] \in [0,1] \wedge r[i-1] < r[i]),$$

$$r[i] := \sum_{j=1}^{i} P(\text{char}(j)); \qquad (i = 0, \ldots d)$$

Definiáljunk egy három értékű ϕ rekurzív függvényt. Az első komponense adja meg az éppen aktuális intervallum alsó, a második komponens pedig a felső végpontját. A harmadik komponens legyen egyenlő az eddig kódolt sorozat hosszával. Ennek alapján:

$$\phi: \boldsymbol{N_0} \rightarrow \boldsymbol{Q} \times \boldsymbol{Q} \times \boldsymbol{N_0}$$

$$\phi(0) := (0,1,0)$$

$$\begin{aligned}\phi(i+1) := \ & (\phi_1(i) + (\phi_2(i) - \phi_1(i)) * r[\text{ord}(s_{i+1}) - 1], \\ & \phi_1(i) + (\phi_2(i) - \phi_1(i)) * r[\text{ord}(s_{i+1})], \\ & \phi_3(i) + 1)\end{aligned}$$

Az eddigiek felhasználásával a kódolás specifikációja:

$$A = S \times R \times N_0 \times Q \quad (s\ r\ n\ l) \qquad S = seq(H)$$

$$B = S \times R \quad (s'\ r')$$

$$Q = (s = s' \wedge r = r')$$

$$R = (r = r' \wedge n = \phi_3(s'.dom) \wedge l = \phi_1(s'.dom))$$

Ezek után a kódoló program a ϕ rekurzív függvény kiszámítása. Alkalmazva az erre vonatkozó tételt úgy, hogy ϕ_1-nek l-et, ϕ_2-nek h-t és ϕ_3-nak n-et feleltetjük meg, kapjuk az alábbi programot:

l,h,n := 0,1,0
ss,ds,s: read
ss
l,h,n := l+(h–l)*r[ord(ds)–1],l+(h–l)*r[ord(ds)],n+1
ss,ds,s: read

A dekódoláshoz vezessük be a ψ három komponensű rekurzív függvényt, illetve az η függvényt. η egy adott kódról döntse el, hogy az melyik karakternek megfelelő intervallumba esik egy adott részintervallum esetén. ψ első két komponense adja meg az aktuális intervallum alsó és felső határát (hasonlóan ϕ-hez), a harmadik komponens pedig legyen az eddig dekódolt sorozat. Tehát:

$\mu: Q \times Q \times Q \rightarrow H$, és

$\mu(v,l,h) = j \iff r[j-1] \leq (v-l)/(h-l) < r[j]$.

$\psi: N_0 \rightarrow Q \times Q \times S$

$\psi(0) := (0,1,\emptyset)$

$$\psi(i+1) := (\psi_1(i)+(\psi_2(i)-\psi_1(i))*r[\mu(v,\psi_1(i),\psi_2(i))-1],$$
$$\psi_1(i)+(\psi_2(i)-\psi_1(i))*r[\mu(v,\psi_1(i),\psi_2(i))],$$
$$con(\psi_3(i),char(\mu(v,\psi_1(i),\psi_2(i)))))).$$

Ezt felhasználva a dekódoló program specifikációja:

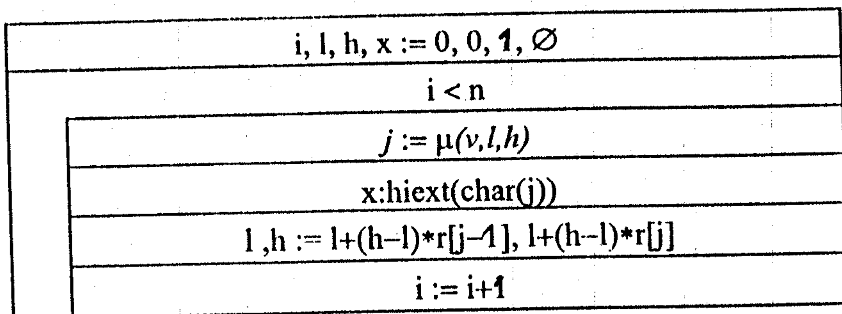
$$A = S \times R \times N_0 \times Q \quad (x\ r\ n\ v)$$

$$B = R \times N_0 \times Q \quad (r'\ n'\ v')$$

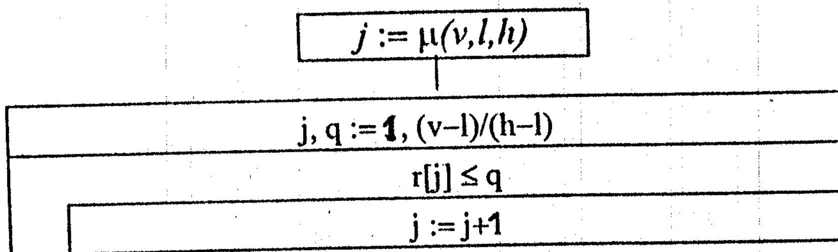
$$Q = (r = r' \wedge v = v' \wedge n = n' \wedge v \in [0,1))$$

$$R = (Q \wedge x = \psi_3(n))$$

A dekódolás programja az előzőhöz hasonlóan visszavezethető a rekurzív függvény kiszámításának tételére. A programban ψ_1-nek l-et, ψ_2-nek h-t és ψ_3-nak x-et feleltettük meg.

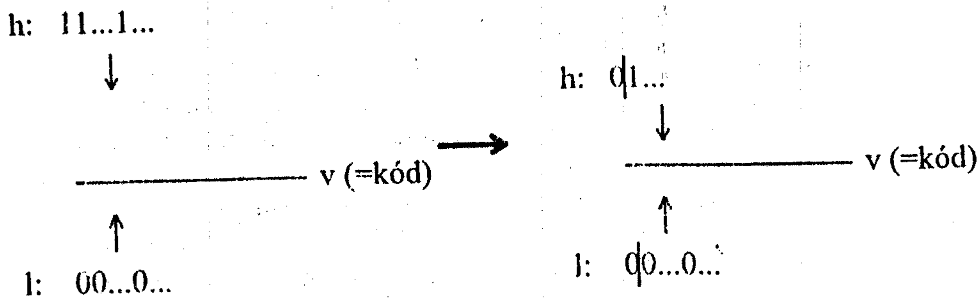


A j := μ(v,l,h) értékadást megvalósító program visszavezethető a lineáris keresés 2. változatára:



Ezzel a problémát elméletileg megoldottnak tekinthetjük. A gyakorlatban azonban ez a megoldás nem lesz megfelelő, ugyanis csak akkor működne jól, ha tetszőleges pontossággal tudnánk ábrázolni a racionális számokat. Erre azért van szükség, mert az intervallumok egyre kisebbek lesznek, azaz a végpontok egyre közelebb kerülnek egymáshoz. Ez azt jelenti, hogy még viszonylag rövid üzenetek esetén is sok tizedesjegy megegyezik a számok elején. Ennek eredménye, hogy amikor a gépi pontosság határára jutunk a kódolás értelmét veszti, hiszen az intervallum egy számmá fajul, és ezt hiába osztanánk fel újra meg újra.

Tegyük fel, hogy csak valahány (pl.: 32) bites egész számokat tudunk ábrázolni, és alakítsuk át a fenti algoritmust ennek megfelelően. Ez egyrészt azt jelenti, hogy nem relatív gyakoriság értékekkel, hanem gyakoriság értékekkel dolgozunk. Ennek megfelelően az r vektor nem a kummulált relatív gyakoriságokat, hanem a kummulált gyakoriságokat tartalmazza. Másrészt az intervallum végpontjait is egészekként ábrázoljuk. Ezen egész számok értéke természetesen jóval meghaladja az ábrázolhatóság határát. Azonban ahogy az intervallumok finomodnak, úgy kerül ezek értéke egyre közelebb egymáshoz, ami azt jelenti, hogy a legmagasabb helyiértékű bitek (az első bitek) értéke megegyezik (ld. ábra).



Ezeket az egyező biteket ki lehet írni, hiszen ezek a későbbiek során sem változnak. A kiírás után pedig ezek elfelejthetők, azaz kiléptethetők, és újabb, alacsonyabb helyiértékű bitek vehetők a számokhoz. A programban nem az [l,h) intervallumot ábrázoljuk, hiszen erre nincs is lehetőség, hanem a [l,h]-t.

Jelölje *F* az ábrázoláshoz használt legnagyobb szám felét, *N* a negyedét, *H* pedig a háromnegyedét. (Azaz F-ben az első bit 1 a többi 0, N-ben a második bit 1 a többi 0, míg H-ban az első két bit 1 a többi 0.) Tudjuk, hogy l és h az intervallum végpontjait ábrázolja, ezért $l \le h$. Ezért ha $h < F$ teljesül, akkor mind l mind h első bitje 0, és ez a későbbiek során sem változik, tehát ez a bit kiírható, kiléptethető és új bit léptethető be alulról mind l-be mind h-ba. Hasonlóan, ha $F \le l$ teljesül, akkor l és h első bitje egy és ez kiírható, és új bit léptethető be. A belépő bit l esetén mindig 0 ($l = 2*l$), h esetén mindig 1 ($h = 2*h+1$), ugyanis így tudjuk a zárt intervallumot ábrázolni.

l és h értéke másként is közel kerülhet egymáshoz, nemcsak a fenti két esetben, és ekkor is újabb bit(ek) bevonására van szükség, annak érdekében, hogy kellően pontos legyen az ábrázolás. Nevezetesen minden olyan esetben amikor a $h-l < F$ feltétel teljesül. Ez teljesül a fenti két esetben, valamint akkor, amikor mind l mind h 'nagyon' megközelíti F-et. Pontosabban, ha $N \le l < F \le h < H$. Ekkor lehetséges, hogy két különböző jel hatására ugyanazt az új l és h értékeket kapnánk, mert nem elég 'finom' az ábrázolás. Ezért új bit bevonásásra van szükség, de ezt nem tehetjük meg olyan egyszerűen mint az előzőekben, hiszen az első bitek különböznek. Az első bit ebben az esetben nem írható ki, hiszen nem tudjuk, hogy a finomított intervallum F melyik oldalára fog esni.

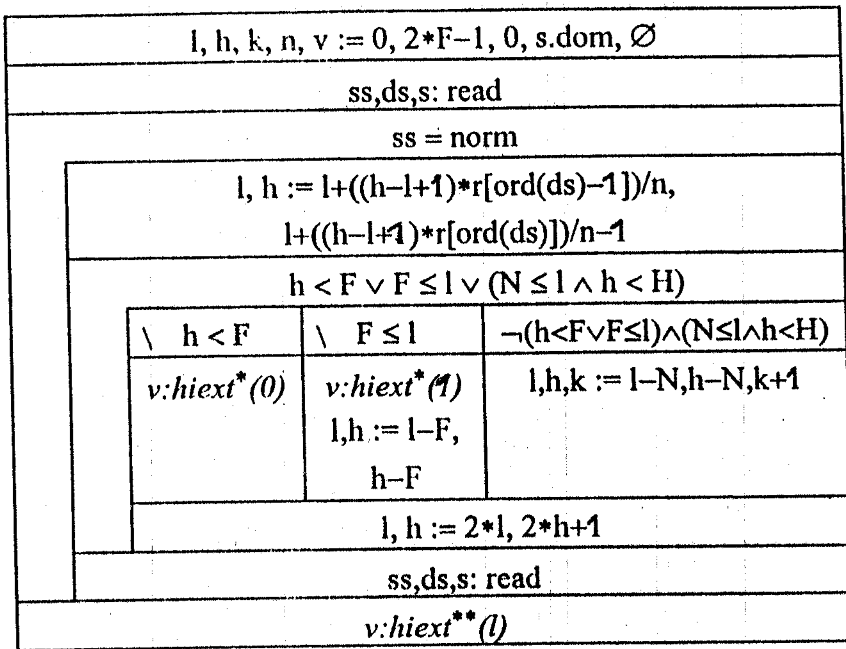
Vizsgáljuk meg részletesebben mit is jelent a $N \le l < F \le h < H$ feltétel. Egyrészt, amint már tárgyaltuk l első bitje 0 h-é pedig 1. Másrészt a 'szélső' egyenlőtlenségek miatt l második (és még utána valahány darab) bitje 1, h második bitje (és még valahány) 0. Azaz l és h bitképe az alábbi:

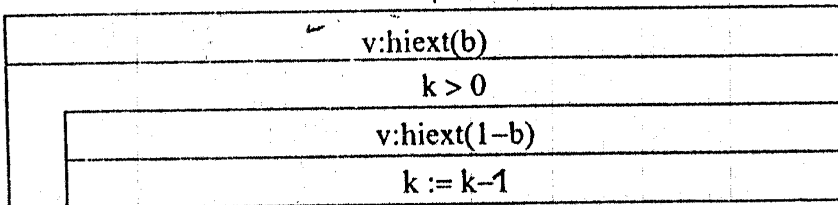
h	1	0	0	...	*b*	...
l	0	1	1	...	*b*	...

Az első bit ugyan különböző l-ben és h-ban, azonban az utána következő valahány bit is különböző, és még azt is tudjuk, hogy ezek különböznek a megfelelő első bitektől. Ez lehetőséget ad arra, hogy az első bit elküldését késleltessük, és csak akkor írjuk ki amikor az értékét már ismerjük. Nevezetesen, az első bitbeli különbséget csúsztassuk át a második bitre, hiszen arra nincs szükségünk, mivel az az első ellentetje. Ez megtehető, ha mindkét számból N-et levonunk. Ezután az első bitet léptessük ki és jegyezzük meg, hogy egy bit 'van a pufferben'. (Vastagított rész jelöli a kiléptetett bitet.)

h	**0**	1	0	...	*b*	...
l	**0**	0	1	...	*b*	...

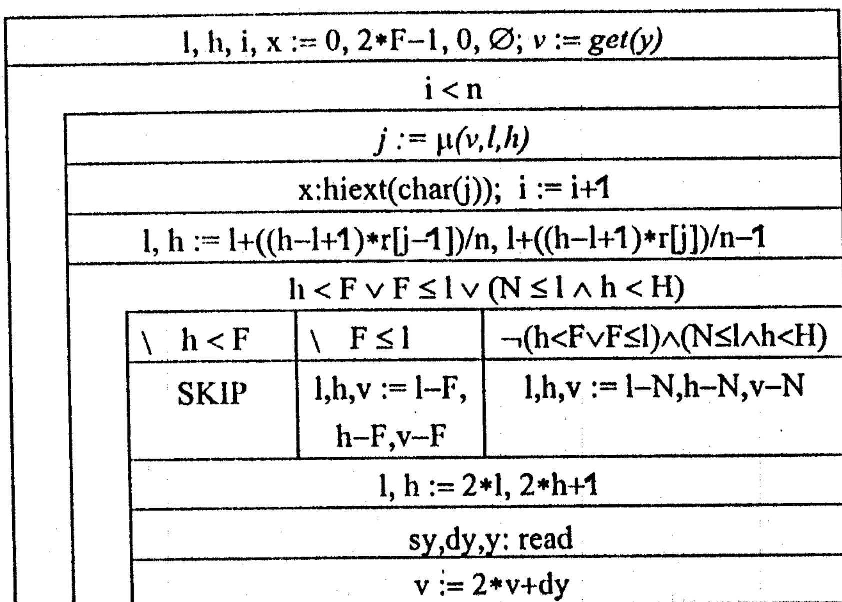
Amikor olyan állapotba jutunk, amelyben már eldönthető, hogy F melyik oldalán van az intervallum, akkor kiírhatjuk az első bitet (az eredeti bitet), és utána megfelelő számú (puffer) ellentétes bitet. Ezt végezze el $v{:}hiext^{*}(b)$. Ezek után a program, ahol *k* jelöli a puffer aktuális méretét, *n* a szöveg hosszát és *v* a kódnak megfelelő bitsorozat.

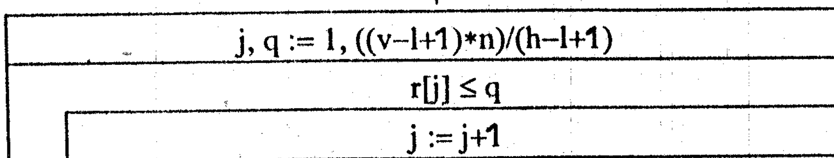


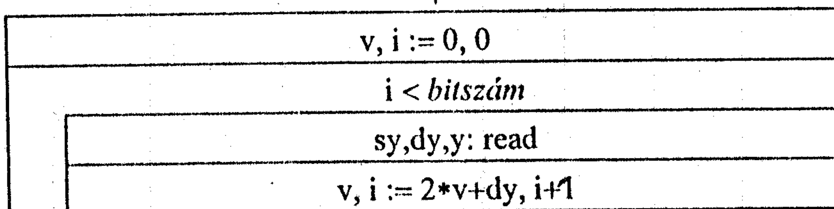


A v:hiext**(l) programrész az l bitjeit írja ki úgy, hogy kezdetben ellenőrzi N és l viszonyát a helyes puffer kezelés érdekében. Ennek megírása az olvasó feladata.

A dekódolás programjának a szerkezete nagyon hasonlít a kódoláséra, ahogy az az elméleti programra is igaz volt, hiszen ennek során az intervallumok analóg módon változnak. Most csak arra kell ügyelnünk, hogy a bináris kód megfelelő részét, *v*-t, is karban kell tartani, hasonlóan az intervallum végpontokhoz. Jelölje *y* a kódolt bitsorozatot *v := get(y)* pedig állítsa elő v-t bitenként teljesen. Azaz y-ból olvasson be az ábrázoláshoz használt bitek számával megegyező számú bitet, és ennek a bináris számnak az értékét rendelje v-hez. Ezek után a program:







Megjegyzés: A programokban szerepel a kettővel való szorzás. Ezt célszerű helyettesíteni balra léptetéssel (<<1 C-ben), és ekkor az F-fel való csökkentés is elhagyható, hiszen a léptetés esetén nincs túlcsordulás. (Egyes fordítóprogramok eleve léptetéssel helyettesítik a kétszerezést. Ha ez nem így lenne, akkor célszerű a számhoz önmagát adni a szorzás helyett.)

Összehasonlítva az aritmetikai tömörítést a Huffman kóddal, megállapítható, hogy az aritmetikai tömörítés eredményezhet rövidebb kódot. Ennek az oka, hogy a Huffman algoritmus egész számú bitet rendel minden egyes jelhez, az aritmetikai tömörírés pedig részbiteket is rendelhet jelekhez, hiszen az egész szöveghez rendel egy kódszót. (Azaz, ha egy jelnek megfelelő optimális bitsorozat hossza 2.6 lenne, akkor a Huffman algoritmus várhatóan 3 bitet használ fel.)

A Huffman algoritmus előnye az aritmetikai tömörítéssel szemben, hogy egy jel kódolásához nem használ fel ismereteket sem az előző, sem a következő jelekről. Az ilyen típusú tömörítéseket *elemenkénti tömörítés*nek nevezzük.

Az aritmetikai tömörítés ugyan nem elemenkénti tömörítés, azonban rendelkezik egy szintén fontos tulajdonsággal. Nevezetesen az input sorozat egyszeri végig olvasásával folyamatosan állítja elő a kódot, aminek kezdőszeletei dekódolhatók. Az ilyen tömörítéseket *on-line tömörítés*nek nevezzük. Természetesen, minden elemenkénti tömörítés on-line tömörítés is egyben.

Az eddigiek során feltettük, hogy a jelek relatív gyakorisága ismert. Gyakorlatban ez sok esetben nem tehető fel. Ilyen esetekben szokás úgynevezett *adaptív modell*eket használni, amelyekben a relatív gyakoriságot egy kezdeti értékből (egyenletes eloszlás) kiindulva a szöveg kódolása során karbantartják. Aritmetikai tömörítés esetén ez megoldható úgy, hogy kezdetben az összes jel gyakorisága 1, és egy jel olvasásakor annak gyakoriságát eggyel növeljük. Mindig az aktuális gyakoriság értékek alapján végezzük el az intervallum újra felosztását. Ez a dekódolás során is megtehető, hiszen a kezdeti intervallum felosztás ismert, és minden egyes jel dekódolásakor a gyakoriság táblázat megfelelően módosítható. Ezt az eljárást a következő algoritmusban részletesen bemutatjuk.

Gráf tömörítés

Rendeljünk egy irányított gráfot az input szöveghez a következőképpen. A gráf csúcsai legyenek az ábécé jelei, az élek feleljenek meg a szövegben egymás után előforduló karakterpároknak. Azaz az x jelű csúcsból vezessen él az y jelű csúcsba, ha a szövegben y x-et követi valahol, vagyis 'xy' előfordul.

Egy lehetséges tömörítési eljárás a fenti gráf használatával az alábbi. Sorszámozzuk meg a csúcsokból kiinduló éleket, minden csúcsra külön-külön. Ekkor egy csúcs esetén elég a megfelelő él sorszámát megadni, hogy a szövegben előforduló következő jelet leírjuk. Nevezetesen annak az élnek a sorszáma adandó meg ami a következő jelet reprezentáló csúcsba vezet. Ha ilyen él még nincs a gráfban, akkor adjuk meg a 0 sorszámot valamint a következő jelet, és bővítsük a gráfot a megfelelő éllel. A dekódolás során a gráf ugyanúgy felépíthető a kódsorozat alapján, mint azt a kódoláskor megtettük. (Mindkét esetben az induláskor a gráf élhalmaza üres.) Tehát a gráfot nem kell a kódsorozattal együtt elküldeni.

Módosítsuk a fenti eljárást úgy, hogy súlyokat rendelünk az élekhez aszerint, hogy azok hányszor voltak használva. (Azaz az él súlya a megfelelő jelpár eddigi előfordulásainak száma, gyakorisága.) A kódsorozat álljon két részből. Az egyikben az él sorszámok sorozatát adjuk meg aritmetikai kódolással, felhasználva a gyakoriság értékeket. A másik sorozat legyen az új élek behúzásához szükséges jelek sorozata. (Esetleg ez is tömöríthető valamilyen módon.)

A kódsorozat második része (a jelek sorozata) elhagyható, abban az esetben, ha a gráfban már kezdetben minden lehetséges élt felveszünk. (Azaz dinamikus gráf ábrázolás helyett statikusra térünk át.) Ez ugyan a gráf ábrázolásához szükséges memória terület méretét jelentősen megnövelheti, azonban a gráfot nem kell a kódsorozattal együtt megadni, így a kód méretét nem növeli. (Az feltehető, hogy mind a kódoló, mind a dekódoló program rendelkezik kellő nagyságú memóriával.) A gráf

éleinek súlya kezdetben legyen 1, később pedig változzanak a gyakoriság értékeknek megfelelően.

Ekkor a kód az élsorozat aritmetikai kódja lesz, amit hasonlóan kapunk mint az adaptív modellt használó aritmetikai kódolás esetén. Ebben az esetben azonban az intervallum felosztás alapját nem a jelek szövegbeli (globális) gyakorisága szabja meg, hanem egy másik jelhez viszonyított (lokális) gyakorisága.

Az egy csúcsból kiinduló éleket ábrázoljuk egy vektorral (*R*), amelyben a kummulált gyakoriságok szerepelnek (ugyanúgy mint az aritmetikai kódolás esetén). A gráfot (*G*) ábrázoljuk a csúcsok vektoraként, azaz a gráf egy vektor a kummulált gyakoriságok vektoraiból (mátrix). Így a megfelelő típusok:

$$G = \text{vekt}([1..d], \mathbf{R});$$
$$R = \text{vekt}([0..d], \mathbf{N});$$
$$I_R(r) = (r[0] = 0 \land \forall i \in [0..r.dom-1]: r[i] < r[i+1]).$$

Az aritmetikai kódolásban használt intervallum finomítást adja meg az α függvény. Ez egy intervallumhoz határozza meg egy jelnek megfelelő új intervallumot a kummulált gyakoriságok és az eddigi hossz (*normálási tényező*) ismeretében. Felhasználva az aritmetikai kódolásnál megismert módszert, ez formálisan:

$$\alpha: \mathbf{Q} \times \mathbf{Q} \times \mathbf{R} \times H \times \mathbf{N} \to \mathbf{Q} \times \mathbf{Q}$$
$$\alpha(l,h,r,e,n) = (\, l+(h-l)*r[ord(e)-1]/n,\ l+(h-l)*r[ord(e)]/n\,).$$

Vezessük be a Γ függvényt, ami egy sorozathoz egy gráfot rendel a fent leírtak szerint, azaz:

$$\Gamma: \mathbf{S} \to \mathbf{G}$$
$$\Gamma(s) = g, \quad \text{ahol:}$$
$$\forall x,y \in H: g[ord(x)][ord(y)] = \sum_{j=1}^{ord(y)} (1 + \sum_{i=1}^{s.dom-1} \chi(s_i = x \land s_{i+1} = char(j)))$$
$$\land\ g[ord(x)][0] = 0.$$

Ez a függvény megadható rekurzívan is. Ennek érdekében vezessünk be két speciális *G* típusú mátrixot, I_0-t és $E_{i,j}$-t. G_0 legyen az a mátrix amelynek minden sorának j. eleme j. $E_{i,j}$ mátrix minden eleme legyen 0, kivéve az i. sorának j. és annál nagyobb indexű elemeit, amelyek értéke legyen 1. Tehát:

$$\forall i \in [1..d]: \forall j \in [0..d]\ G_0[i][j] = j.$$

$$\forall k \in [1..d] \setminus \{i\}: \forall l \in [0..d]: E_{i,j}[k][l] = 0 \land$$
$$\forall l \in [0..j-1]: E_{i,j}[i][l] = 0 \land \forall l \in [j..d]: E_{i,j}[i][l] = 1.$$

Ezek segítségével Γ rekurzív kifejtése:

$\Gamma(\emptyset) = G_0$;

$\Gamma(<e>) = G_0$;

$\Gamma(\text{hiext}(s,e)) = \Gamma(s)+E_{\text{ord}(s.\text{hiv}),\text{ord}(e)}$.

A gráf-aritmetikai tömörítés ami egy szöveghez egy intervallumot rendel, megadható az alábbi *gra* rekurzív függvénnyel:

$\text{gra}: \boldsymbol{S} \rightarrow \boldsymbol{Q}\text{x}\boldsymbol{Q}$

$\text{gra}(\emptyset) = [0,1)$;

$\text{gra}(<e>) = [\ (\text{ord}(e)-1)/d,\ \text{ord}(e)/d\ )$;

$\text{gra}(\text{hiext}(s,e)) = \alpha(\ \text{gra}(s), \Gamma(s)[\text{ord}(s.\text{hiv})], e, \Gamma(s)[\text{ord}(s.\text{hiv})][d]\ )$.

Itt a definíció utolsó sorában azt mondjuk meg, hogy egy egy elemmel hosszabb sorozatra a függvény az eredeti sorozat által megadott intervallumot (gra(s)) a megfelelő kummulált gyakoriságok (Γ(s)[ord(s.hiv)]) és az új jel (e) alapján skálázza át. Γ(s)[ord(s.hiv)] tartalmazza a számunkra szükséges kummulált gyakoriságokat, hiszen s.hiv az utolsó jel, azaz a gráf ennek megfelelő csúcsában vagyunk, és az innen kivezető élek gyakoriságai alapján kell az új intervallumot meghatározni. A normálási tényező pedig Γ(s)[ord(s.hiv)][d], hiszen a megfelelő kummulált gyakoriságok tömb utolsó eleme adja meg, hogy eddig összesen hányszor szerepelt a jel.

A gra függvényt felhasználva a gráf-aritmetikai tömörítés specifikációja (feltesszük, hogy a tömörítendő sorozat hossza legalább 2):

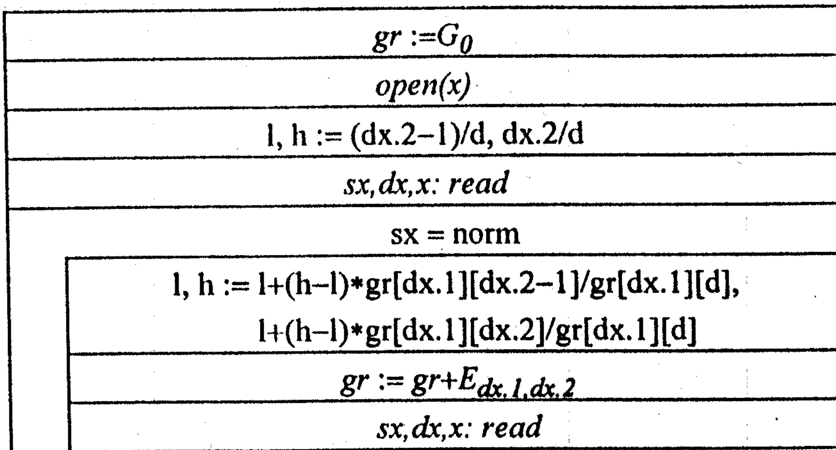
$\mathbf{A} = \underset{s}{\boldsymbol{S}}\text{x}\underset{l}{\boldsymbol{Q}}\text{x}\underset{h}{\boldsymbol{Q}}$ ($\boldsymbol{S} = \text{seq}(H)$)

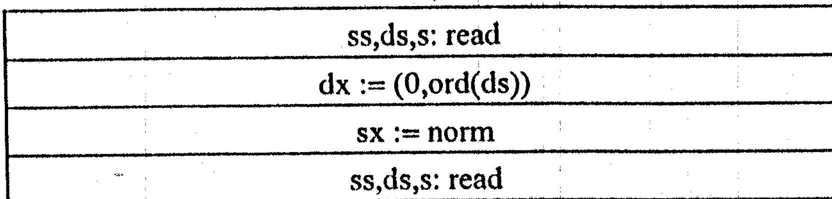
$\mathbf{B} = \underset{s'}{\boldsymbol{S}}$

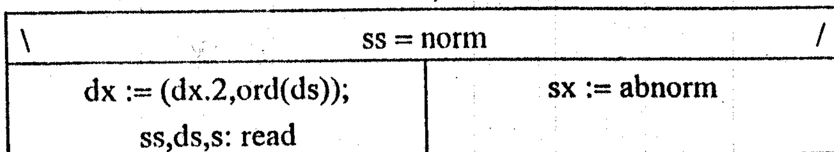
$\mathbf{Q} = (s = s' \wedge 2 \leq s.\text{dom})$

$\mathbf{R} = (\ (l,h) = \text{gra}(s')\ )$.

A feladat megoldását visszavezetjük rekurzív függvény értékének kiszámítására. A gra függvény értékeit az *l*, *h* változókkal, a Γ függvény értékét a *gr* mátrix változóval helyettesítsük. A feladatban szereplő ***S*** típusú sorozatról térjünk át egy párokból álló sorozatra, ahol a pár első tagja az ord függvény értéke az ***S***-beli sorozat megelőző jelére, második tagja az ord értéke az aktuális jelnek. Ezek alapján a program:







A gr mátrix változóra vonatkozó nem megengedett értékadások kitranszformálása az olvasó feladata. (Ezek egyszerű ciklusok.)

A fenti program gyakorlatban ugyanúgy használhatatlan, mint az aritmetikai tömörítés esetén a megfelelő program. Annak érdekében, hogy használható megoldáshoz jussunk ugyanazt a reprezentációt kell választani, amit ott használtunk, és a programot az ott megismerttel analóg módon kell átalakítani. (Ezt az olvasóra hagyjuk.)

Ziv-Lempel tömörítés

A Ziv-Lempel algoritmus alapötlete, hogy az ismétlődő részsorozatok helyett, azok megelőző előfordulásának helyét és hosszát írjuk ki. Természetesen a hatékonyabb tömörítés érdekében a lehető leghosszabb egyezést válasszuk ki. Tehát a létrehozandó kód számpárokból áll: az előző előfordulás helye, (*p* - pozíció) és mérete (*m*). Ha nincs előző előfordulás, akkor írjuk ki az adott pozíción található jel sorszámát és nullát. (Nulla nem lehet egy egyezés mérete, így ez jó megkülönböztetés.)

A tömörítendő szöveget most vektornak tekintjük, ugyanis az algoritmusban többször át kell olvasni. (Lehetne olyan sorozat is ahol az indexelés megengedett, de a gyakorlatban ez vektort jelent.) Legyen ***T*** a szöveg, ***X*** a kódsorozat, ***K*** a kódpár típusa. Ekkor:

$T = \text{vekt}(N,\text{H});$

$X = \text{seq}(K),$ ahol: $K = (\text{p}:N, \text{m}:N_0)$

Jelölje egy t vektor esetén $t_{a..b}$ a vektor a indexénél kezdődő és b-nél végződő részsorozatát (részvektorát), azaz

$t_{a..b} := < t[a], t[a+1], \ldots, t[b] >.$

Ez értelemszerűen üres, ha b < a. Vezessük be a 3 változós *hol* függvényt, ami megadja, hogy a szöveg egy adott pozíciójától (i) kezdődő megadott méretű (m) rész hol szerepel a szövegben a pozíciót megelőzően, illetve nullát ad ha ilyen előfordulás nincs. Tehát:

$\text{hol}: T \text{x} N \text{x} N \rightarrow N_0$

$\text{hol}(t,i,m) := \min\{ j \mid t_{i..i+m-1} = t_{j..j+m-1} \},$ ha $\exists j<i: t_{i..i+m-1}=t_{j..j+m-1},$

$\text{hol}(t,i,m) := 0$ különben

A hol függvény segítségével definiálható a szöveg egy pozíciójához ***K*** típusú értéket előállító *kód* függvény.

$\text{kód}: T \text{x} N \rightarrow N \text{x} N_0$

$\text{kód}(t,i) := (\text{ord}(t[i]),0)$ ha $\text{hol}(t,i,1) = 0,$

$\text{kód}(t,i) := (p,m)$ ha $p = \text{hol}(t,i,m) \wedge m>0 \wedge \text{hol}(t,i,m+1) = 0.$

Tehát, ha az i. pozíción új jel van (hol(t,i,1)=0), akkor nincs előző előfordulás, így a jel sorszámát és nullát adunk meg; különben egy előző előfordulás helyét (p=hol(t,i,m)) és hosszát úgy, hogy ennél hosszabb előfordulás ne legyen (hol(t,i,m+1) = 0). Így a kód függvényt a megfelelő pozíciókra kiszámolva és ezeket az értékeket kiírva megkapjuk a tömörített szöveget.

A pozíciók kiszámításához tudnunk kell, hogy az eddig kiszámított párok, azaz az eddigi kódsorozat, a szövegből hány jelet reprezentál. Ekkor a következő pár előállításához a kód függvényt kell kiszámolni a szöveg következő jelére. Vezessük be a *hossz* függvényt, ami megadja egy K típusú értékekkel definiált sorozat által kódolt eredeti szöveg hosszát. Ezt a függvényt rekurzívan definiáljuk:

$\text{hossz}: X \rightarrow N_0$

$\text{hossz}(\varnothing) := 0,$

$\text{hossz}(\text{hiext}(x,k)) := \text{hossz}(x)+\max(1,k.m).$

Üres sorozat természetesen 0 hosszúságú üzenetet kódol. Egy új kódpár hozzávételével pedig legalább eggyel nő a reprezentált szöveg hossza, hiszen ha k.m=0, akkor k.p a jel sorszáma; különben k.m méretű részsorozatot ír le k.

Az eddigiek felhasználásával a tömörítés specifikációja:

$A = T \mathrm{x} X$
$\quad t \quad x$

$B = T$
$\quad t'$

$Q = (t = t')$

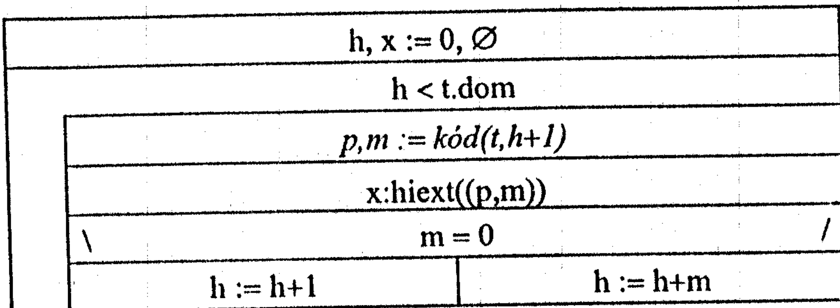
$R = (Q \wedge \mathrm{hossz}(x) = t.\mathrm{dom} \wedge \forall i \in [1..x.\mathrm{dom}]: x_i = \mathrm{kód}(t, \mathrm{hossz}(x_{1..i-1})+1))$

Az utófeltétel $x_i = \mathrm{kód}(t,\mathrm{hossz}(x_{1..i-1})+1)$ része garantálja, hogy a kódsorozat soron következő eleme az előző i–1 pár által reprezentált rész ($\mathrm{hossz}(x_{1..i-1})$) utáni résznek megfelelő kódpárt adja meg. A hossz(x) = t.dom jelentése, hogy az x kódsorozat az egész szöveget reprezentálja.

A feladatot ciklussal lehet megoldani, amelynek invariánsa:

$$P = (Q \wedge \forall i \in [1..x.\mathrm{dom}]: x_i = \mathrm{kód}(t,\mathrm{hossz}(x_{1..i-1})+1)),$$

terminátor függvénye pedig: $t.\mathrm{dom} - \mathrm{hossz}(x_{1..i})$. Helyettesítsük a hossz rekurzív függvényt a *h* változóval, és transzformáljuk ki a függvényt. Így a program (aminek levezetése az olvasó feladata):



A fenti programban a p,m := kód(t,h+1) értékadás nem megengedett, így azt meg kell valósítani. Ennek egy lehetséges módja lenne a kód függvény kiszámítását visszavezetni egy lineáris keresésre, ahol az első olyan m értéket keressük, amire hol(t,i,m) = 0. A hol függvény kiszámítása pedig egy string keresési feladat, amire már ismerünk algoritmusokat. Ennek a megoldásnak a hátránya, hogy igen nagy a művelet igénye, azaz nem hatékony.

Próbáljuk meg másképp megvalósítani a kód függvényt. Definiáljuk az *egyezik* függvényt, ami megadja a szöveg két pozíciójára az azoktól kezdődő leghosszabb megegyező szakasz méretét.

$$\mathrm{egyezik}: T \mathrm{x} N \mathrm{x} N \rightarrow N_0$$
$$\mathrm{egyezik}(t,i,j) := \max\{ m \in N_0 \mid t_{i..i+m-1} = t_{j..j+m-1} \}.$$

A fenti definícióval nyilván ekvivalens:

$$egyezik(t,i,j) := \min\{ m \in N_0 \mid t[i+m] \neq t[j+m] \}.$$

Ennek segítségével felírható a kód függvény az alábbi formában:

$$kód(t,i) := (ord(t[i]),0) \quad \text{ha} \quad \max\{ egyezik(t,i,j) \mid j < i \} = 0,$$

$$kód(t,i) := (p,m) \quad \text{ha} \quad m = \max\{ egyezik(t,i,j) \mid j < i \} \wedge m = egyezik(t,i,p) \wedge m > 0.$$

Ezt felhasználva a kód függvény értékei kiszámolhatók maximum kereséssel, elkerülve az előző megoldásban meglevő ismételt összehasonlításokat. A maximum keresésben szereplő egyezik függvény kiszámolható lineáris kereséssel. Ez így még mindig elég lassú programhoz vezetne. Szerencsére az egyezik függvény értéke a legtöbb helyen nulla. Ezek az értékek elhagyhatók a maximum keresésből, azaz kevesebb értékből kell maximumot kiválasztani, amivel javítható a program hatékonysága.

Szűkítsük le a maximum keresést olyan elemekre, ahol az egyezik függvény értéke legalább 2. Ez megtehető, miután nulla esetén a jel és 0 lesz a kód, és 1 esetén szintén ki lehet írni a jelet és egy nullát, hiszen ezzel nem lesz rosszabb a tömörítés. (Eredetileg egy előző előfordulás helye és 1 került volna ki.) Ez persze azt jelenti, hogy eltértünk az eredeti kód függvénytől. (Az olvasó feladata a kód függvény megfelelő módosítása.)

A leszűkítés érdekében vezessük be a *pár* függvényt, ami egy adott jelpárra megadja, hogy a szövegben milyen pozíciókon fordul az elő. Ekkor a maximum keresést a pár függvény i-nél kisebb értékeire kell elvégeznünk.

$$pár: T x H x H \rightarrow set(N)$$

$$pár(t,x,y) := \{ i \in N \mid t[i] = x \wedge t[i+1] = y \}.$$

Valósítsuk meg a pár függvényt úgy, hogy az értékként felvett halmazokat listákkal ábrázoljuk, és a listák kezdetére egy mátrix elemei mutatnak. Azaz a pár(t,x,y) halmaznak megfelelő lista első elemét a *pár* mátrix ord(x) sorának ord(y) oszlopába eső eleme adja meg.

A halmazok elemeit (illetve a listákat) ábrázoljuk egy közös vektorban, legyen ez *lista*. A vektor indextartománya (hossza) egyezzen meg a szöveg indextartományával (hosszával). A vektor elemei tartalmazzák a megfelelő halmaz következő elemét, ami egyben a rákövetkező elem helyének vektorbeli indexe is legyen, illetve legyen nulla, ha ez az utolsó elem a halmaznak megfelelő listában. Tehát, ha pár(t,x,y)={100,20,5}, akkor pár[ord(x),ord(y)]=100, lista[100]=20, lista[20]=5, lista[5]=0 legyen. A lista vektorban nem léphet fel ütközés a különböző halmazok között, hiszen a szöveg egy pozícióján csak egy jelpár kezdődhet.

Tehát a függvényt megvalósító mátrix és vektor:

$$pár = mátrix([1..d],[1..d],N_0);$$

$$lista = vekt(N,N_0).$$

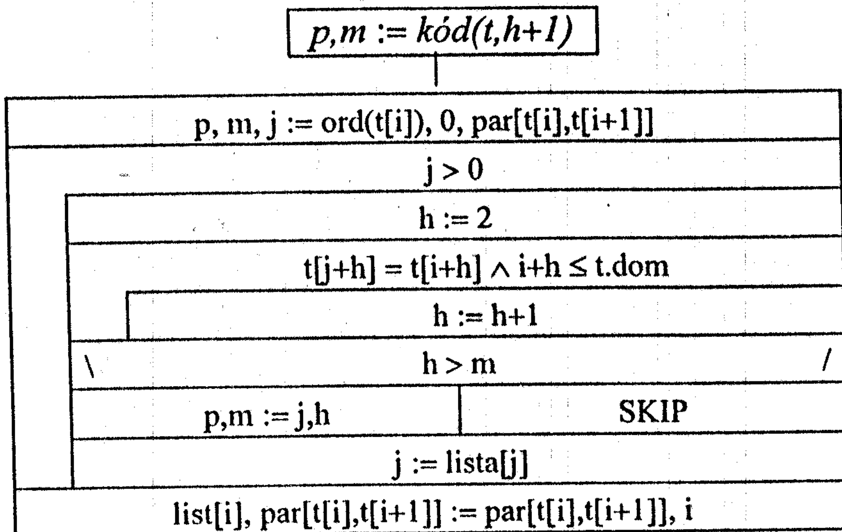
Négy műveletet értelmezünk a fenti kettőssel leírt típuson:

- init: feltöltés üres halmazokkal;
- put: egy elem berakása a párnak megfelelő halmazba;
- first: egy halmaz első eleme;
- next: egy halmaz következő eleme.

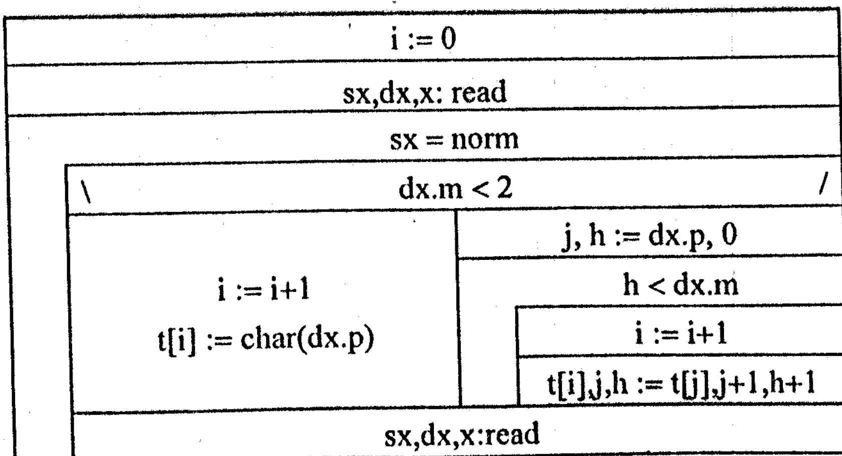
A formális típusspecifikáció, illetve típus megadása az olvasó feladata. Az egyes műveletek megvalósítása az alábbi:

- init: A pár mátrix és a lista vektor feltöltése nullákkal.
- put(x,y,i): list[i],par[ord(x),ord(y)] := par[ord(x),ord(y)],i;
- j := first(x,y): j := par[ord(x),ord(y)];
- k := next(j): k := lista[j].

Az init műveletet a főprogram elején kell végrehajtani, a kód függvény kiszámítása pedig a fentiek felhasználásával a következő:



A dekódolás programja értelemszerűen:



Ziv-Lempel-Welch tömörítés

Az algoritmus alapötlete, hogy a szövegben előforduló részsorozatokat vegyük fel az ábécébe új jelként. Ezt egy kódtáblázat segítségével valósítjuk meg. Ez a táblázat kezdetben csak az ábécé jeleit tartalmazza. Ezt bővítjük ki lépésenként úgy, hogy a szövegben felismert leghosszabb táblázatban szereplő kódot kiegészítjük a szövegben utána következő jellel. Így a táblázat elemei egyre hosszabb jelsorozatokat írnak le. A tömörített szöveg pedig a táblázat maga lesz. Ahhoz, hogy ez tényleg tömörítés legyen ügyesen kell ezt a táblázatot megadnunk. Először egy módosított változatot adunk meg.

Egy táblázatbeli bejegyzés álljon két részből, az első legyen egy nem negatív egész szám, a második pedig egy ábécébeli (eredeti H-beli) jel. A nem negatív egész szám adja meg, hogy a kód melyik előző kódból kapható meg úgy, hogy ahhoz hozzáfűzzük a jelet; azaz ez egy táblázatbeli index. A táblázat elején legyenek az ábécé jelei, és az ezekhez tartozó első komponens legyen 0, ami azt jelenti, hogy ezekhez nem tartozik megelőző kód.

Tekintsük a következő példát. H legyen egy három betűs ábécé, nevezetesen H:={a,b,c}, a szöveg pedig: abcabcabcabcabc... .Ekkor a táblázat első három sora rendre a (0,a), (0,b), (0,c) bejegyzéseket tartalmazza. A szöveg tömörítésekor elolvassuk az 'a' jelet, ami szerepel a táblázatban, de a következő jel hozzávételével keletkező 'ab' már nem. Ezt tehát új bejegyzésként felvesszük. Ez a kódszó az első kódszóból ('a') és a 'b' jelből épül fel. Így a megfelelő bejegyzés (1,b) lesz. A szöveg következő jele a 'c'. Ez is benne van a táblázatban, de ha a következő jelet is hozzávasszük 'ca'-t kapjuk, ami új. Vegyük fel ezt új bejegyzésként, azaz a táblázat 5. eleme legyen (3,a). Ezután a hatodik elem (2,b) lesz. Ezután az 'ab' jelsorozatot megtaláljuk a táblázatban, de 'abc'-t már nem, így új bejegyzésként bekerül: (4,c). Ekkor 'abc' már szerepel, de 'abca' még nem, tehát felvesszük a (7,a)-t. A táblázat eleje tehát:

1	0	a
2	0	b
3	0	c
4	1	b
5	3	a
6	2	c
7	4	c
8	7	a
9	6	a
...	...	...

A táblázatot (*T*) ábrázoljuk egy vektorral, amelynek elemei párok. Minden pár első komponense egy nem negatív egész szám, ami megfelel a megelőző kódrésznek, a második komponens pedig a konkatenálandó jel. A szöveget (*S*) tekintsük most egy sorozatnak. Így a megfelelő típusok:

$S = \text{seq}(H);$

$T = \text{vekt}(N,K)$, ahol: $K = (\text{kód}:N_0, \text{jel}:H);$

$I_T(t) = \forall i \in [1..t.\text{dom}]: t[i].\text{kód} < i.$

Az invariáns tulajdonság azt a természetes követelményt fejezi ki, hogy a táblázatban egy kód csak őt megelőzőre hivatkozhat.

Mielőtt továbbmennénk vizsgáljuk meg, hogy egy táblázatbeli bejegyzéshez hogyan tudjuk meghatározni az általa kódolt szöveget. Vezessük be ennek érdekében az *str* függvényt, amit rekurzív módon definiálunk. Ez értelemszerűen a táblázatbeli bejegyzés jel részéből álló egy hosszúságú sorozat, ha a kód rész nulla. Egyébként a kódrész által megadott bejegyzésnek megfelelő sorozathoz kell a jelet hozzávenni. Formálisan:

$\text{str}: N \rightarrow S$

$\text{str}(k) := < t[k].\text{jel} >$ ha $t[k].\text{kód} = 0;$

$\text{str}(k) := \text{con}(\text{str}(t[k].\text{kód}), < t[k].\text{jel} >)$ ha $t[k].\text{kód} > 0.$

Vezessük be továbbá a *hossz* függvényt, ami megadja, hogy a táblázat első *j* eleme milyen hosszúságú szöveget kódol. Az első d jel az ábécét írja le, így a kódolt szöveg hossza a d+1 és j közé eső bejegyzések által kódolt szövegek hosszának összege. Tehát:

$\text{hossz}: N \rightarrow N$

$$\text{hossz}(j) := \sum_{k=d+1}^{j} (\text{str}(k).\text{dom}).$$

A kódolás specifikációja során szükség lesz egy függvényre, ami a sorozat elejéből levágja a táblázat által már kódolt részt. Ezért definiáljuk a *vág* függvényt, ami egy sorozat elejéről elhagy megadott számú jelet.

$\text{vág}: S \times N \rightarrow S$

$\text{vág}(s,n) := \text{lorem}^n(s).$

Vizsgáljuk meg, hogy egy adott sorozat és táblázat esetén hogyan határozható meg a táblázat végére kerülő új pár. Meg kell keresni a táblázatban azt a kódot ami a szöveg lehető leghosszabb kezdőszeletét adja meg, és ennek indexe lesz a pár első komponense. Ezt a kezdőszeletet levágva a szövegből a maradék első jele lesz a pár második komponense. Vezessük be a *zlw* függvényt, ami meghatározza, hogy a táblázat melyik kódja illeszkedik leghosszabban a szöveg elejére, és megadja a maradék sorozatot.

$\text{zlw}: S \rightarrow N \times S$

$\text{zlw}(s) := (k,x)$: $k = \max\{ j \in N \mid \exists x \in S: s = \text{con}(\text{str}(j),x)\}.$

(A fenti definíció elégséges számunkra, azonban pontosan meg is tudjuk határozni a benne szereplő x sorozatot, hiszen azt úgy kaphatjuk, hogy s-ből str(k).dom hosszú kezdőszeletet levágunk, azaz:

$$zlw(s) := (k, vág(s, str(k).dom)) : k = \max\{j \mid s = con(str(j), vág(s, str(j).dom))\}).$$

A táblázatba új bejegyzésként a (k,x.lov) pár kerül be. Ezeket felhasználva a kódolás specifikációja:

$$A = \underset{s}{\boldsymbol{S}} \times \underset{t}{\boldsymbol{T}}$$

$$B = \underset{s'}{\boldsymbol{S}}$$

$$Q = (s = s')$$

$$R = (\forall i \in [1..d]: t[i] = (0, char(i)) \wedge$$
$$\forall i \in [d+1..t.dom]: t[i].kód = zlw_1(vág(s', hossz(i-1))) \wedge$$
$$t[i].jel = zlw_2(vág(s', hossz(i-1))).lov \wedge hossz(t.dom) = s'.dom.$$

A feladat egy megoldható egy szekvenciával, amelynek az első tagja a táblázat első d elemét tölti fel a specifikációnak megfelelően (legyen ez az *init* programrész). A második rész egy ciklus, ami elemenként állítja elő a táblázat elemeit. Ennek alapján a program, aminek levezetését a megadott **P** invariáns felhasználásával az olvasóra bízzuk:

$$P = (\forall i \in [1..d]: t[i] = (0, char(i)) \wedge j \in [d..t.dom] \wedge$$
$$\forall i \in [d+1..j]: t[i].kód = zlw_1(vág(s', hossz(i-1))) \wedge$$
$$t[i].jel = zlw_2(vág(s', hossz(i-1))).lov.$$

```
init(t)
i := d;
s ≠ Ø
    k,x := zlw(s)
    t[i+1] := (k,x.lov)
    s := lorem(x)
    i := i+1
```

A fenti programban a zlw függvény nem megengedett, így azt meg kell valósítanunk. Ezt megtehetnénk egy feltételes maximum kereséssel a definíció alapján. Ekkor azonban minden egyes lépésben ki kellene számolni az str függvény értékét. Ennek az eljárásnak a műveletigénye nagy (szöveg hosszával négyzetes arányban is állhat), ezért próbáljunk meg hatékonyabb megoldást találni.

Tegyük fel, hogy már találtunk egy j kódot úgy, hogy str(j) illeszkedik a szöveg kezdőszeletére. Ekkor j-nél nagyobb (hosszabb) kód csak úgy lehet prefixe a szövegnek, ha a kódnak str(j) a prefixe. A táblázat felépítése miatt ez azt jelenti, hogy

a hosszabb kód egyrészt j után szerepel a táblázatban, másrészt hivatkozik j-re. Ez a hivatkozás lehet direkt, ha csak egy jellel különböznek, illetve indirekt, ami azt jelenti, hogy j valahol előfordul a kódok láncában ha azt visszafejtjük.

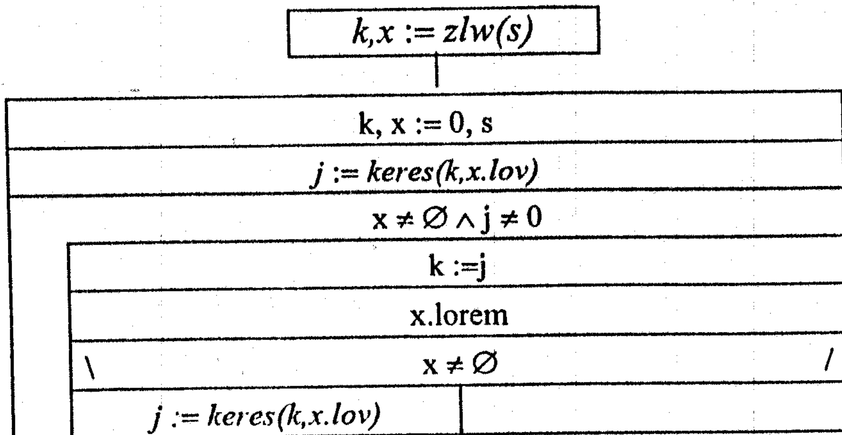
Ezt a tulajdonságot kihasználva a zlw függvényt megvalósíthatjuk egy lineáris kereséssel, amelyben a prefixet állítjuk elő egészen addig, amíg a prefix növelhető. Ennek érdekében vezessük be a *keres* függvényt ami egy táblázatbeli bejegyzés helyét adja meg, ha van ilyen bejegyzés, illetve nullát, ha ez nem fordul elő. Tehát:

keres: $\mathcal{N} \times H \to \mathcal{N}_0$

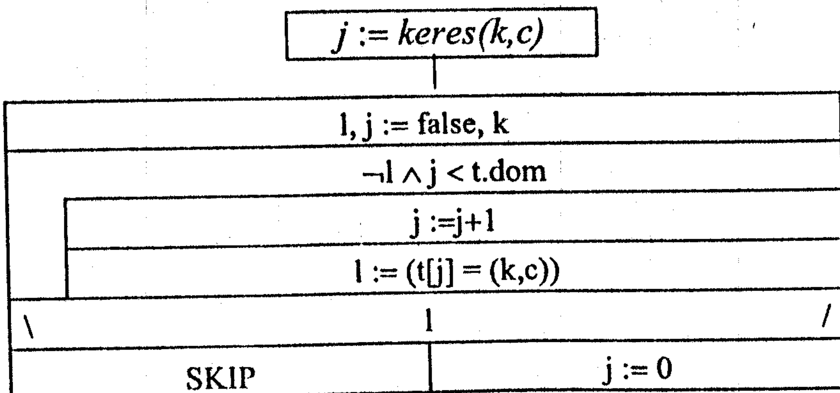
keres(k,c) := 0 ha $\forall i \in [1..t.dom]: t[i] \neq (k,c)$;

keres(k,c) := j ha $t[j] = (k,c)$.

A keres függvény értékein kell keresni a (0,x.lov) párból kiindulva úgy, hogy a következő pár első komponensének a függvény értékét, második komponensének pedig x következő jelét választjuk. Ezt egészen addig kell folytatni, amíg keres értéke 0 nem lesz (és x nem üres), ugyanis ekkor a prefix tovább már nem növelhető. Ennek alapján a program:



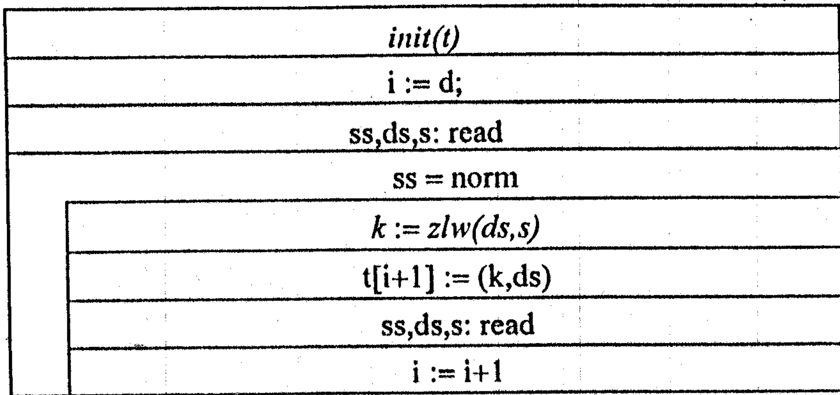
A *j := keres(k,x.lov)* értékadás visszavezethető egy lineáris keresésre a [k..t.dom] intervallumon:

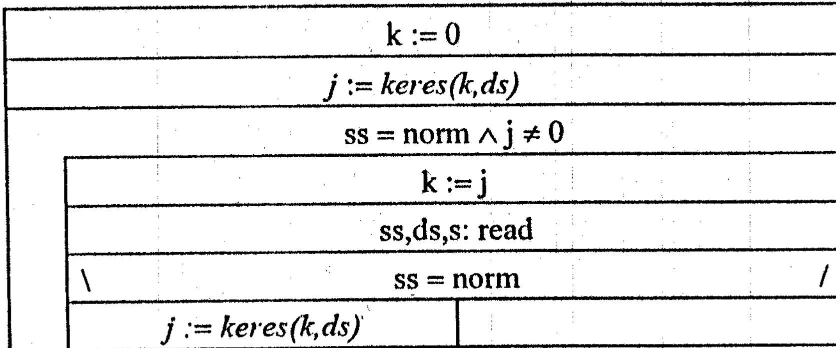


A keres függvény hatékonyabban megvalósítható, ha sikerül egy jó hash függvényt találni. Ennek a hash függvénynek egy táblázatbeli bejegyzéshez (kód-jel pár) kell egy

természetes számot rendelnie. Ennek könnyen és gyorsan számolhatónak kell lennie, valamint jól kell szórnia, azaz különböző párokra lehetőleg különböző eredményt kell adnia. A hash függvény segítségével a lineáris keresés kiküszöbölhető, hiszen az minden elemhez egy kulcsot rendel, aminek alapján az megtalálható.

A megoldó programunkban elég szabadon kezeltük a szövegeket, hiszen ezeket értékül adtuk egymásnak (k,x := zlw(s) illetve x := s). Ha jobban megvizsgáljuk a megoldást, láthatjuk, hogy erre nincs is szükség, elegendő mindig egy elemet olvasni a sorozatból. Ennek megfelelően a főprogramot illetve a k,x := zlw(s) programot átírva szekvenciális input fájlra az x sorozat kiinvertálásával kapjuk az alábbi megoldást:





Dekódoláskor a táblázatbeli kódoknak megfelelő str függvény értékeket kell kiszámolni a táblázat d+1. elemétől kezdődően, és ezeket a sorozatokat kell konkatenálni. A program specifikációja:

$$A = \underset{s}{S} \times \underset{t}{T}$$

$$B = \underset{t'}{T}$$

$$Q = (t = t')$$

$$R = (Q \wedge s = \underset{i=d+1}{\overset{t.dom}{\mathrm{con}}}(str(t[i].kód)).$$

A megoldó program egy ciklus, ami egy elemenkénti feldolgozás a táblázat d+1..t.dom elemein. Ebben az str függvény értéke kiszámolható egy verem segítségével. A program megírása (az str által előállított sorozat kiinvertálásával) az olvasó feladata.

Az eddigiekben leírt módszert tovább lehet fejleszteni, és így jutunk el a valódi Ziv-Lempel-Welch tömörítéshez. Nevezetesen, elegendő a táblázat első oszlopát megadnunk, ha a táblázat felépítését egy kicsit módosítjuk. Követeljük meg, hogy a táblázatbeli kódok ne a szöveg egymás utáni diszjunkt szakaszait ábrázolják, hanem legyen köztük egy jel átfedés. Azaz a következő kód első jele egyezzen meg az előző kód utolsó jelével. (Ez persze ahhoz vezet, hogy a kódszavak rövidebb részeket írnak le, azonban ezt várhatóan ellensúlyozza az, hogy csak egy oszlop a kód.)

Vizsgáljuk meg miként módosul a táblázat a bemutatott példában. A táblázat elején az ábécé jelei szerepelnek, mint eddig. A 4. sor felel meg az 'ab' sorozatnak, akárcsak az előbb. Az 5. sor azonban nem a 'ca' sorozatot kódolja, mint eddig, hanem a 'bc'-t, hiszen egy jel átfedést követelünk meg. Ezek után a 6. sor felel meg a 'ca'-nak, a 7. pedig 'abc'-nek, ami a 4. sorból és a 'c' jelből épül fel. Így a táblázat:

1	0	a
2	0	b
3	0	c
4	1	b
5	2	c
6	3	a
7	4	c
8	6	b
9	5	a
...	...	...

A kódoláshoz használjuk fel a táblázat ábécének megfelelő részéből (első d sor, most első 3 sor), a jeleket, a további részekből pedig az első oszlopot, illetve az utolsó sorból a jelet. Ennek alapján a dekódolás elvégezhető, hiszen vegyük a d+1. elemet (most 4.) és nézzük meg, hogy ez milyen jelre hivatkozik. Ez lesz az első jel. A kódszó utolsó jelével most ne törődjünk ezt megkapjuk a következő kódszó elején. Nézzük a következő bejegyzést, és fejtsük vissza a neki megfelelő kódot (str függvény értéke), az utolsó jeltől eltekintve. Konkatenáljuk ezt az eddigi sorozathoz. Folytassuk ezt, amíg az utolsó sorhoz nem érünk. Itt a megfelelő kód után még írjuk ki az adott jelet is, és megkaptuk az eredeti szöveget.

Tegyük fel, hogy a szöveg 'abcabcabca' volt. Ekkor a táblázat utolsó bejegyzése a 9. sor lesz. Végezzük el a dekódolást a fentiek szerint. Ekkor a 4. sor alapján kapjuk,

hogy az első jel 'a'. Az 5. és 6. sorokból megkapjuk 'b'-t és 'c'-t. Eddig tehát 'abc'-t dekódoltuk, valamint kiegészítettük a táblázat 4. és 5. sorát a 'b' és a 'c' jelekkel. A 7. sornak megfelelő kódszó (az utolsó jel nélkül) 'ab', ezt hozzávesszük a szöveghez ('abcab'), és az előző 6. sort kiegészítjük az 'a' jellel. Ezt folytatva a 9. sor után eljutunk az 'abcabcabc' szöveghez, amihez hozzávéve az utolsó 'a' jelet megkapjuk az eredeti üzenetet.

A már megadott specifikáció és függvények könnyen módosíthatók, hogy ezt az utóbbi tömörítést írják le. Ez az olvasó feladata. A kódoló program módosítása csak abból áll, hogy a főprogramban a ciklusmagban szereplő ss,ds,s: read -et elhagyjuk. (Pontosan ez lépte át a kódszó által leírt utolsó jelet.) A dekódoló program egyetlen módosítása (a táblázat második oszlopának felépítésén kívül), az utolsó elem hozzáírása a szöveghez.

Ziv-Lempel-Welch-Zsoldos tömörítés

Az algoritmus hasonló az előző tömörítéshez, csak most a táblázatbeli új kód nem egy régi kódból és egy jelből épül fel, hanem két régi kód konkatenációjaként áll elő. Ennek eredménye, hogy a kódok hossza jóval gyorsabban nőhet. Ez persze azt jelenti, hogy egy táblázatbeli bejegyzés nem egy kódból és egy jelből, hanem két kódból fog állni. A táblázat elején továbbra is az ábécé jelei legyenek, és az ezekhez tartozó első komponens legyen 0, ami azt jelenti, hogy ezekhez nem tartozik megelőző kód. (A második komponens legyen a jel sorszáma, hiszen most ennek is egy számnak kell lennie.)

Nézzük, hogy milyen táblázatot rendel ez a módszer az abcabcabcabc... szöveghez, ha H={a,b,c}. Az első három bejegyzés megfelel az ábécé jeleinek. A negyedik bejegyzés az 'ab' jelsorozatot kódolja, ami az első bejegyzésnek megfelelő kód (a) és a második bejegyzésnek megfelelő kód (b) konkatenációja. Az ötödik bejegyzés a 'cab' sorozatot kódolja mint a 3. (c) és 4. (ab) kódok konkatenációja. A hatodik bejegyzés a 'cabcab' sorozatnak felel meg, amit az 5. kód duplázásával kapunk meg. A hetedik bejegyzés a 6. bejegyzés duplázásával áll elő, feltéve, hogy a szöveg még elég hosszú. Így a táblázat:

1	0	a
2	0	b
3	0	c
4	1	2
5	3	4
6	5	5
7	6	6
...	...	...

Az előzőekhez hasonlóan a táblázat reprezentálására használjunk vektort, aminek az invariánsa hasonlít a már megismert invariánsra, csak most mind a két kód komponensre igaz, hogy az csak megelőző lehet. Tehát:

$S = \text{seq}(H)$;

$T = \text{vekt}(N,K)$, ahol: $K = (\text{kód}_1{:}N_0, \text{kód}_2{:}N_0)$;

$I_T(t) = \forall i \in [d+1..t.\text{dom}]: t[i].\text{kód}_1 < i \wedge t[i].\text{kód}_2 < i.$

Vezessük be az előzőekhez hasonlóan az *str*, *zlwzs* függvényeket, amelyek hasonlítnak a már megismert str és zlw függvényekhez, csak most kétszeres rekurzió szerepel bennük. (A zlwzs definiálásához felhasználjuk a zlw függvényt.) A *hossz* és *vág* függvény pedig megegyezik a már megismerttel, csak a hossz függvényben az új str függvényt kell használni. (A zlw függvény formálisan ugyanaz mint eddig, azonban a jelentése különböző hiszen a benne szereplő str függvény megváltozott.)

$\text{str}: N \rightarrow S$

$\text{str}(k) := \text{char}(k)$ ha $k \le d$;

$\text{str}(k) := \text{con}(\text{str}(t[k].\text{kód}_1), \text{str}(t[k].\text{kód}_2))$ ha $k > d$.

$\text{hossz}: N \rightarrow N$

$$\text{hossz}(j) := \sum_{k=d+1}^{j} (\text{str}(k).\text{dom}).$$

$\text{zlwzs}: S \rightarrow N \times N \times S$

$\text{zlwzs}(s) := (k_1,k_2,x)$ ahol:

$k_1 = \text{zlw}_1(s) \wedge (k_2,x) = \text{zlw}(\text{vág}(s,\text{str}(k_1).\text{dom})).$

Ezeket felhasználva a kódolás specifikációja:

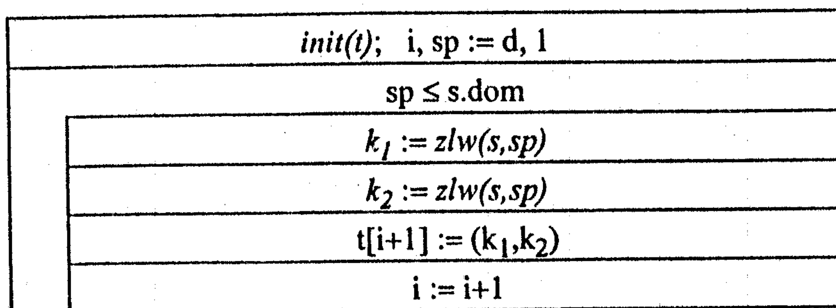
$A = S \times T$
$\quad s \quad t$

$B = S$
$\quad s'$

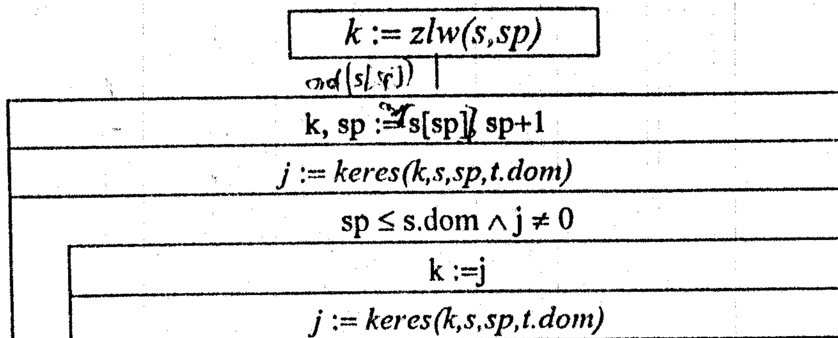
$Q = (s = s')$

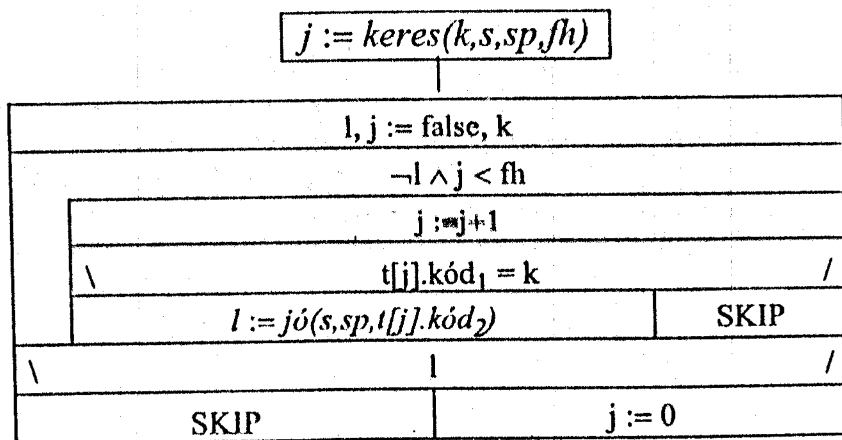
$R = (\forall i \in [1..d]: t[i] = (0,i) \wedge \text{hossz}(t.\text{dom}) = s'.\text{dom} \wedge$
$\quad \forall i \in [d+1..t.\text{dom}]: t[i] = \text{zlwzs}_{1,2}(\text{vág}(s',\text{hossz}(i-1)))).$

A megoldó program megegyezik a Ziv-Lempel-Welch tömörítésnél megismerttel, csak a zlw függvény helyett a zlwzs függvény értékét kell kiszámolni. A zlwzs függvény kiszámítása pedig megegyezik zlw függvény értékének kétszeri kiszámításával. (Azaz $k_1,k_2 := \text{zlwzs}(ds,s)$ helyett a $k_1 := \text{zlw}(ds,s)$; $k_2 := \text{zlw}(ds,s)$ szekvencia írható.) A zlw függvény kiszámításához cseréljük az S sorozat típust vektorra. Vezessük be az *sp* változót, ami megadja, hogy a szövegből hány jelet kódoltunk eddig. Így a program:

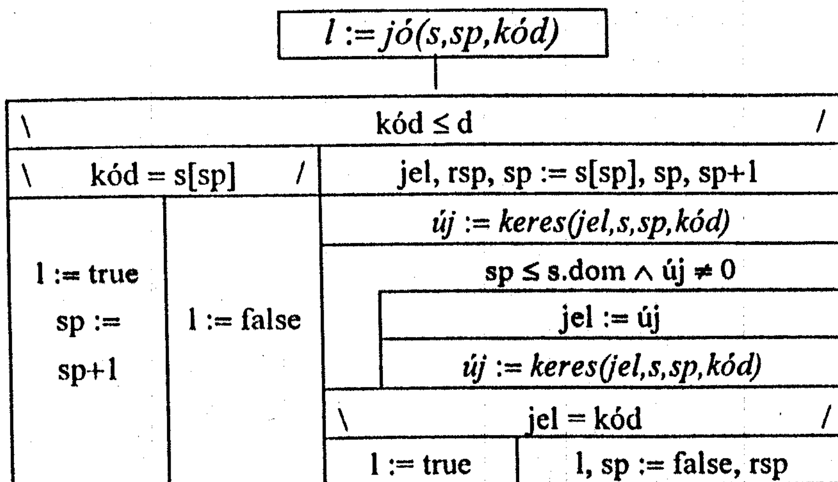


A zlw függvény kiszámítása hasonlít a Ziv-Lempel-Welch algoritmusnál bemutatott megoldáshoz, azaz megint a kódok prefixumait kell kihasználni, hogy kiküszöböljük az str függvény kiszámítását. Ebben az esetben azonban figyelembe kell venni, hogy a táblázat második bejegyzése is kód. Ez azt jelenti, hogy a keres függvény bonyolultabb lesz, nevezetesen ellenőrizni kell, hogy a második kód is illeszkedik a szöveg megfelelő részére. Végezze el ezt az ellenőrzést a *jó* függvény. Ezek alapján a zlw és a keres programok:





A jó függvényt megvalósító program megadható egy elágazással, amelynek feltétele, hogy a kód egy egyszerű ábécébeli jel, illetve összetett kód. Az első esetben (kód≤d) az illeszkedés eldönthető a kód és a szövegbeli jel összehasonlításával; a második esetben a kód illeszkedését kell vizsgálni, ami hasonlít a zlw függvény kiszámítására. Ekkor azonban az adott kódnál tovább nem kell ellenőrizni a táblázatbeli bejegyzéseket, továbbá vissza kell tudnunk lépni a szövegben a kiinduló pozícióra, ha a kód megsem illeszkedik (rsp). Így a program:



A bemutatott program rekurziót tartalmaz, ennek megfelelően nem illeszkedik a programozási módszertanból eddig megismert eszközökhöz. (A rekurzió miatt az egyes függvényeket megvalósító programok argumentumai is paraméterekként kezelendők; a programrészekben szereplő változók lokális változók.) Ez a probléma megoldható a rekurzió feloldásával, amit az 'Adatszerkezetek' tantárgyból ismertnek tekintünk. Ennek elvégzése az olvasó feladata.

A dekódolás megegyezik a Ziv-Lempel-Welch esetén alkalmazott dekódolással, csupán csak az str függvény kiszámítása módosul. A verembe most mindkét kódot be kell tenni egy bejegyzés feldolgozásakor. Így az str függvényt kiszámító program: