ELTE IK - Programtervező Informatikus BSc

Záróvizsga tételek

14. Alapvető algoritmusok

Alapvető algoritmusok és adatszerkezetek

Függvények aszimptotikus viselkedése, algoritmusok hatékonysága. Összehasonlító rendezések (beszúró, összefésülő, gyors- és kupacrendezés), maximális műveletigény alsó korlátja. Rendezés lineáris időben (bucket, leszámláló és radix rendezés). Adattömörítés (naiv, Huffman, LZW). Mintaillesztés (brute-force, quicksearch, KMP).

1 Függvények aszimptotikus viselkedése, algoritmusok hatékonysága

TODO

2 Összehasonlító rendező algoritmusok (buborék és beszúró rendezés, ill. verseny, kupac, gyors és összefésülő rendezés)

Buborék- és beszúró rendezés klasszikusak, n^2-es műveletigényűek, a többi hatékony, $n\log(n)$-es idejűek.

2.1 Buborékrendezés

A legnagyobb értéket cserékkel a végéig felbuborékozza, ezt minden ciklus végén elhagyjuk. A gyakorlatban nem használják.

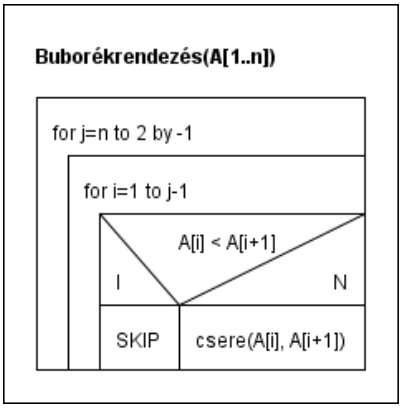


Figure 1: Buborékrendezés

2.2 Beszúró rendezés

Kis n-re (kb 30) ez a rendezés a legjobb.
Itt az elemmozgatás mindig 1 értékadás (buborékrendezésnél a csere 3 értékadás). Listára is implementálni lehet, ez esetben a pointereket állítjuk át, az elemek helyben maradnak.
$A[1..j]$ rendezett, $j = 1..n$.

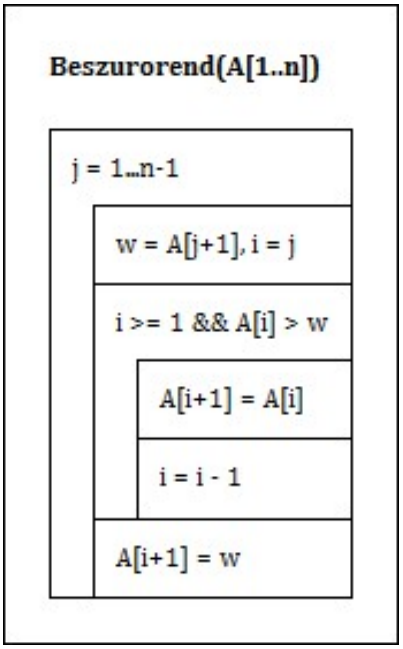


Figure 2: Beszúró rendezés

2.3 Versenyrendezés

Gyakorlatban nem használják.
Teljes bináris fa az alapja, egy versenyfa. Szintfolytonosan ábrázoljuk tömbösen.

1. A versenyfa kitöltése (a verseny lejátszása). Maximum a gyökérben, ennek kiírása az outputra.
2. $(n-1)$-szer
   a) gyökérben szereplő maximális elem helyének megkeresése a levélszinten és $-\infty$ írása a helyére
   b) az egészet újrajátsszuk (azt az ágat, ahol volt) $\rightarrow$ 2. legjobb feljut a gyökérbe

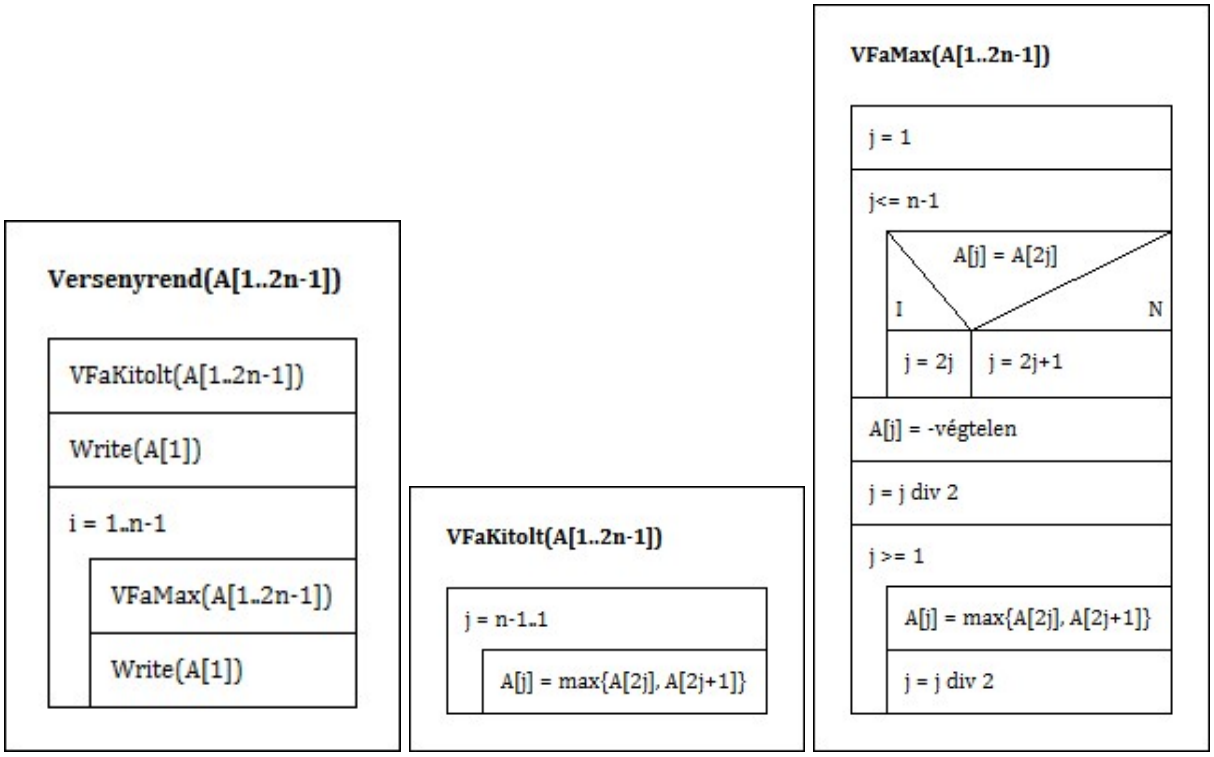


Figure 3: Versenyrendezés

2.4 Kupacrendezés

1. Kezdő kupac kialakítása. Rendezetlen input tömbből tartalmi invariánst készítünk, ami már kupac struktúrájú. Elv: cserékkel lesüllyesztjük az elemet a nagyobb gyerek irányába, ha kisebb a nagyobbik gyereknél. A süllyesztés eljuthat ahhoz a csúcshoz, amelynek nincs jobb gyereke.

2. $(n-1)$-szer

   a) gyökérelem és az alsó szint jobb szélső (=utolsó) aktív elemének cseréje, és a csere után lekerült elem inaktívvá tétele

   b) a gyökérbe került elem süllyesztése az aktív kupacon

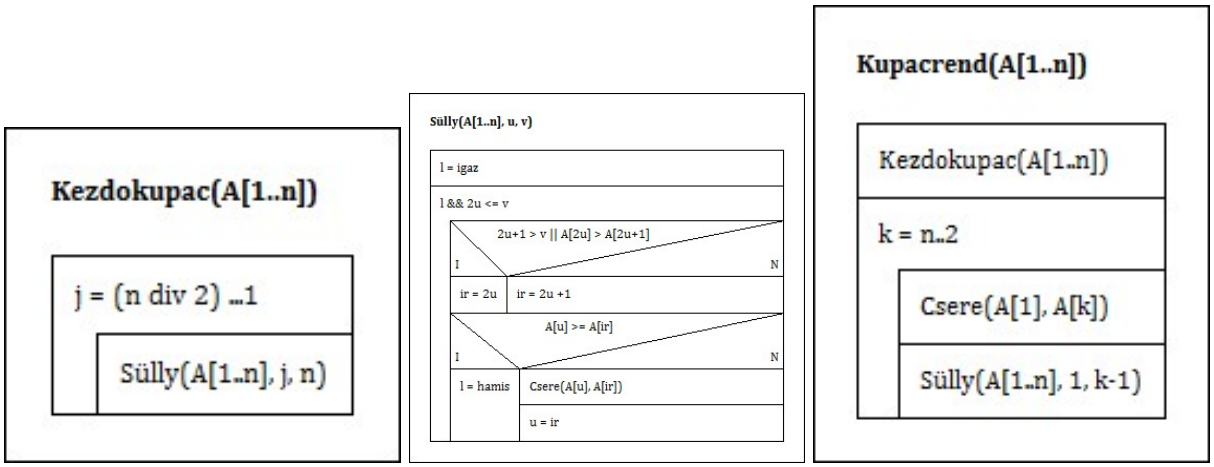


Figure 4: Kupacrendezés

A kezdőkupac kialakításánál, és a ciklus közben a süllyesztés módja kicsit különbözik, hiszen az első esetben a változó elem süllyed le a teljes kupacon, a másodikban a gyökér süllyed az aktív kupacon. A képen látható algoritmus mindkét műveletet teljesíti.

2.5 Gyorsrendezés

Elve: véletlenül választunk egy elemet. A nála kisebb elemeket tőle balra, a nagyobbakat jobbra rakjuk, az elemet berakjuk a két rész közé. Rekurzív algoritmus.

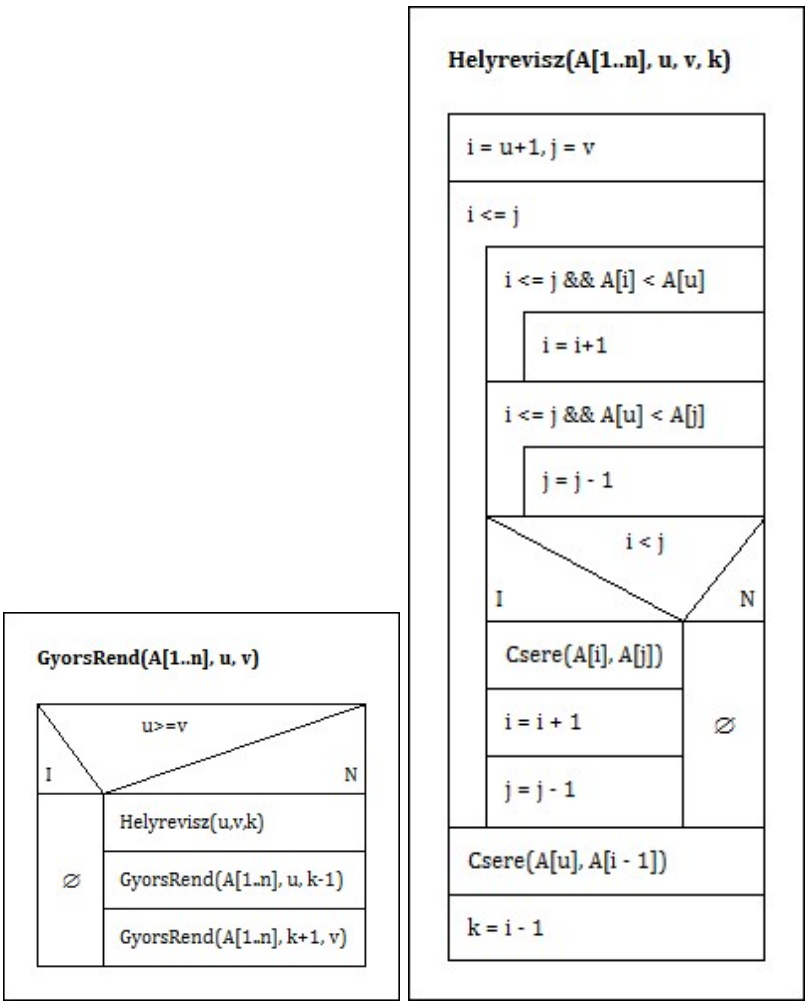


Figure 5: Gyorsrendezés

2.6 Összefésülő rendezés

Alapja: 2 rendezett sorozat összefésülése. Ezt alkalmazhatjuk felülről lefelé (rekurzív) vagy alulról felfelé (iteratív), ez utóbbit szekvenciális fájloknál.

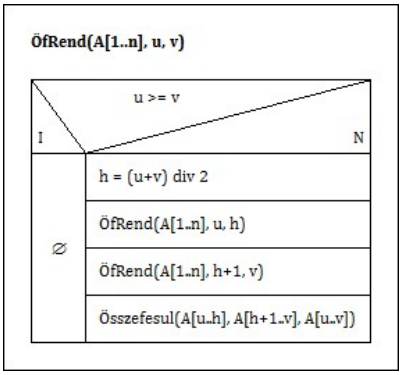


Figure 6: Összefésülő rendezés

3 A műveletigény alsó korlátja összehasonlító rendezésekre

3.1 Műveletigény

Kijelöljük a domináns műveleteket, és az n inputméret függvényében hányszor hajtódnak végre, ezt nézzük. Jelölés általánosan $T(n)$, de lehet konkrétan is, pl $Cs(n)$ [csere]. $mT(n)$ a minimális műveletigény, $MT(n)$ a maximális és $AT(n)$ az átlagos.

Θ : nagyságrendileg azonos, két konstans közé beszorítható

$\mathcal{O}$: nagyságrendi felső becslés, o: nincs megengedve az egyenlőség

Ω : nagyságrendi alsó becslés, ω: nincs megengedve az egyenlőség

3.2 Alsókorlát

Például: n elem maximumkiválasztása legalább $(n-1)$ összehasonlítást igényel. Bizonyítása: Ha ennél kevesebb összehasonlítás lenne, akkor legalább 1 elem kimaradt, és ezzel ellentmondásba kerülhetünk.
Döntési fa: Algoritmus n méretű inputra. Kiegyenesednek a ciklusok véges hosszú lánccá, a végrehajtás nyoma egy fa struktúrát ad. Tökéletes fa: minden belső pontnak 2 gyereke van. Ennél az algoritmusnál nincs jobb, mert $2^{h(t)} \geq n!$, összehasonlító rendezés esetén, $n!$ input.

3.3 Alsókorlát legrosszabb esetben

Tétel: $MO_R(n) = \Omega(n \log n)$ A legkedvezőtlenebb permutációra legalább $n \log n$ összehasonlítás. Bizonyítás: $\log_2 n! \leq n \log_2 n = \Omega(n \log n)$, és $MO_R(n) = h(t) \geq \log_2 n!$ (lemma miatt) $\Rightarrow MO_R(n) = \Omega(n \log n)$.

3.4 Alsókorlát átlagos esetben

Legyen minden input egyformán valószínű ($\frac{1}{n!}$).
$AO_R(n) = \frac{1}{n!} \sum_{p \in Perm(n)} O_R(p)$, és könnyű belátni, hogy $\sum_p O_R(p) = lhsum(h(t_R(n)))$ [levél-magasság-összeg].
Lemma: Az $n!$ levelet tartalmazó tökéletes fák közül azokra a legkisebb az $lhsum(h(t_R(n)))$ érték, amelyek majdnem teljesek.
Tétel: $AO_R(n) = \Omega(n \log n)$.

4 Rendezés lineáris időben (bucket-, leszámláló- és radix rendezés)

4.1 Bucket rendezés

A bucket rendezés egy olyan rendezési algoritmus, amelyet általában egyenletesen elosztott értékekkel dolgozó adatsorok rendezésére használnak.

- A bemeneti adatsorban meghatározunk egy tartományt vagy intervallumot, amelyben az értékek eloszlása közel azonos. Ez a tartomány általában a minimális és maximális érték közötti intervallum.
- Létrehozunk egy vagy több "vödröt" vagy "bucketet", amelyek a tartományon belül helyezkednek el. A vödrök száma lehet állandó vagy változó, és attól függ, hogy milyen módon osztjuk el a tartományt.
- Az adatokat a megfelelő vödrökbe helyezzük a kulcsuk alapján. Ez történhet például a kulcsérték egészrészének kiszámításával vagy a hash függvény segítségével.
- Minden vödörben levő adatsort rendezünk. Ez a rendezési lépés lehet bármely más rendezési algoritmus alkalmazása, például beszúrási rendezés vagy gyorsrendezés.
- Az egyes vödrökből származó rendezett adatsorokat összevonjuk, hogy előálljon a rendezett kimeneti adatsor.

```
bucket_sort( L : list )
n := the length of L
B : list[n] // Create the buckets B[0..n)]
j := 0 to (n−1)
    Let B[j] be empty list
L ≠ ⊘
    Remove the first element of list L
    Insert this element according to its key k into list B[[n ∗ k]]
j := 0 to (n−1)
    Sort list B[j] nondecreasingly
Append lists B[0], B[1], . . . , B[n − 1] in order into list L
```

4.2 Leszámláló rendezés

A leszámláló rendezés egy hatékony és stabil (az azonos értékű elemek sorrendjét megőrző) lineáris időkomplexitású rendezési algoritmus.

- Először meghatározzuk a bemeneti adatsor legnagyobb és legkisebb értékét, hogy meghatározzuk a tartományt, amelyben az értékek eloszlanak. Ez a lépés szükséges ahhoz, hogy létrehozhassunk egy leszámláló tömböt, amely tartalmazza a különböző értékek előfordulási számát.
- Létrehozunk egy leszámláló tömböt, amelynek mérete a tartomány méretével egyezik meg. A leszámláló tömb minden eleme kezdetben 0.
- Végigmegyünk a bemeneti adatsoron, és minden elem előfordulását megszámoljuk a leszámláló tömbben. Az elemeket a leszámláló tömb indexével azonosítjuk.
- A leszámláló tömbben frissítjük az elemek előfordulási számát azzal, hogy hozzáadjuk az előző elem előfordulási számát. Ez a lépés segít abban, hogy meghatározzuk az adott elem végleges helyét a rendezett adatsorban.
- Végigmegyünk újra a bemeneti adatsoron, és minden elemet a leszámláló tömb alapján beszúrunk a rendezett adatsor megfelelő pozíciójába. Eközben a leszámláló tömbből csökkentjük az adott elem előfordulási számát, hogy kezeljük az esetleges azonos értékeket.
- Az eredményül kapott rendezett adatsor a bemeneti adatsor rendezett változata.

```
counting_sort(A, B : T[n] ; r : N ; φ : T → [0..r))
C : N[r] // counter array
k := 0 to r−1
    C[k] := 0 // init the counter array
i := 0 to n − 1
    C[φ(A[i])]++ // count the items with the given key
k := 1 to r−1
    C[k] += C[k − 1] // C[k] := the number of items with key ≤ k
i := n − 1 downto 0
    k := φ(A[i]) // k := the key of A[i]
    C[k] − − // The next one with key k must be put before A[i] where
    B[C[k]] := A[i] //Let A[i] be the last of the {unprocessed items with key k}
```


4.3 Radix rendezés

A radix rendezés egy hatékony, stabil és nem összehasonlító rendezési algoritmus, amelyet általában egész számok rendezésére alkalmaznak.

- Először meghatározzuk a bemeneti adatsor legnagyobb értékét. Ez szükséges ahhoz, hogy meghatározzuk, hány számjegyet kell figyelembe venni a rendezés során.
- Kezdjük a rendezést a legkisebb számjegytől (legkevésbé jelentős) a legnagyobbig (legjelentősebb). Ez lehet az egységjegy, tizedesjegy, százalékjegy stb.
- A bemeneti adatsort rendezzük a jelenlegi számjegy szerint. Ez általában egy stabilitást megőrző rendezési algoritmus, például beszúrási rendezés vagy leszámláló rendezés alkalmazásával történik.
- A rendezett adatsort átrendezzük és elhelyezzük az eredményül kapott sorrendben. Ez a lépés a stabilitást biztosítja, hogy az azonos értékű elemek sorrendje ne változzon.
- Ismételjük meg a 2-4 lépéseket az összes számjegyre, a legkevésbé jelentőstől a legjelentősebbig. Ezáltal az adatsorban a rendezés minden számjegy szerint megtörténik.
- Az eredményül kapott rendezett adatsor a bemeneti adatsor rendezett változata.

Az input (azaz bemeneti) lista, szimbolikus jelöléssel ($r = 4; d = 3$):
$L = \langle 103, 232, 111, 013, 211, 002, 012 \rangle$

1. menet (a számok jobbról 1., azaz jobbszélső számjegyei szerint):
$B_0 = \langle \rangle$
$B_1 = \langle 111, 211 \rangle$
$B_2 = \langle 232, 002, 012 \rangle$
$B_3 = \langle 103, 013 \rangle$
$L = \langle 111, 211, 232, 002, 012, 103, 013 \rangle$

2. menet (a számok jobbról 2., azaz középső számjegyei szerint):
$B_0 = \langle 002, 103 \rangle$
$B_1 = \langle 111, 211, 012, 013 \rangle$
$B_2 = \langle \rangle$
$B_3 = \langle 232 \rangle$
$L = \langle 002, 103, 111, 211, 012, 013, 232 \rangle$

3. menet (a számok jobbról 3., azaz balszélső számjegyei szerint):
$B_0 = \langle 002, 012, 013 \rangle$
$B_1 = \langle 103, 111 \rangle$
$B_2 = \langle 211, 232 \rangle$
$B_3 = \langle \rangle$
$L = \langle 002, 012, 013, 103, 111, 211, 232 \rangle$

Figure 7: Példa, mivel jobban érthető mint az algo

5 Adattömörítések

5.1 Naiv adattömörítés

A tömörítendő szöveget karakterenként, fix hosszúságú bitsorozatokkal kódoljuk. $\sum = \sigma$ 1, σ2, . . . , σi az ábécé.

Egy-egy karakter $\lceil \lg d \rceil$ bittel kódolható, ui. $\lceil \lg d \rceil$ de biten $2^{\lceil \lg d \rceil}$ de különböző bináris kód ábrázolható, és $2^{\lceil \lg d \rceil} \geqslant d > 2^{\lceil \lg d \rceil - 1}$, azaz $\lceil \lg d \rceil$ biten ábrázolható d-féle különböző kód, de eggyel kevesebb biten már nem. $In : \sum\langle\rangle$ a tömörítendő szöveg. n = |In| jelöléssel n * $\lceil \lg d \rceil$ bittel kódolható. Pl. az ABRAKADABRA szövegre d = 5 és n = 11, ahonnét a tömörített kód hossza $11 * \lceil lg5 \rceil = 11 * 3 = 33$ bit. (A 3-bites kódok közül tetszőleges 5 kioszthatóaz 5 betűnek.) A tömörített fájl a kódtáblázatot is tartalmazza. A fenti ABRAKADABRA szöveg kódtáblázata lehet pl. a következő:

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

5.2 Huffman-algoritmus

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

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

Az algoritmus (spec. bináris Huffman fára):

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

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

A	B	C	D	E
5	4	3	2	1

Ekkor a redukciós lépések a következők:

A	B	C	D, E
5	4	3	3
C, D, E	A	B
6	5	4
A , B	C, D, E
9	6
A , B , C, D, E
15

A huffman-fa a XY. ábrán látható.

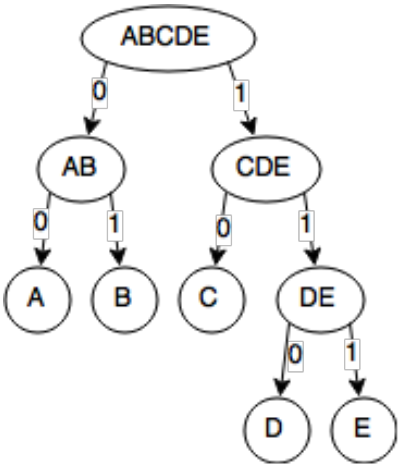


Figure 8: Huffman-fa példa

Tehát a kódszavak:

A	B	C	D	E
00	01	10	110	111

5.3 LZW-algoritmus

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

Kódolás

A kódolás algoritmusa a következő lépésekből áll:

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

Dekódolás

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

Tehát a dekódolás lépései:

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

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

Példa:

Szöveg: AAA
Szótár: A - 1

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

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

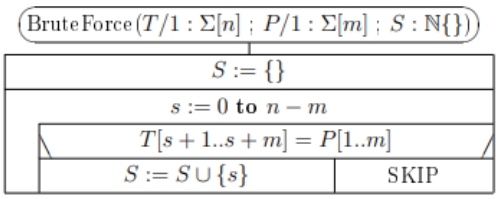
Ebben az esetben a dekódolásnál, egy trükköt vetünk be. A szótárba írás pillanatában még nem ismert a beírandó szó utolsó karaktere (A példában A-t találtuk, de nem volt 2-es bejegyzés). Ekkor ?-et írunk a szótárba írandó szó utolsó karakterének helyére. (Tehát A? - 2 kerül a szótárba). De mostmár tudni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

6 Mintaillesztés

6.1 Brute-force mintaillesztés

A brute force algoritmus egy egyszerű, de nem mindig hatékony módszer, amelyet problémák megoldására alkalmaznak. A brute force megközelítés során az algoritmus minden lehetséges lehetőséget kipróbál a probléma megoldására, majd megtalálja a helyes eredményt.

- Meghatározzuk a probléma lehetséges megoldási területét. Ez lehet egy adott adatsor, egy meghatározott intervallum, vagy egyéb feltétel alapján meghatározott értékek.
- Generáljuk az összes lehetséges kombinációt vagy lehetőséget a megoldási területen. Ez lehet például egy iteráció, amely végigmegy minden lehetséges értéken vagy kombináción.
- Ellenőrizzük minden lehetséges kombináció vagy lehetőség esetén, hogy az adott megoldás kielégíti-e a probléma feltételeit vagy kritériumait.
- Tároljuk el a helyes megoldásokat, vagy válasszuk ki a legjobb eredményt a probléma alapján.



6.2 Knuth-Morris-Pratt algoritmus

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

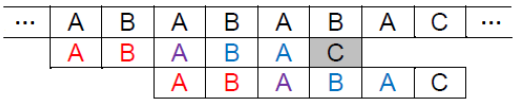


Figure 9: KMP algoritmus több karakter tolás estén

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

Azt, hogy ez egyes esetekben mekkorát tolhatunk nem kell minden elromlás alkalmával vizsgálni. Ha a mintára önmagával lefuttatjuk az algoritmus egy módosított változatát (10. ábra), kitölthetünk egy tömböt, mely alapján a tolásokat végezni fogjuk.

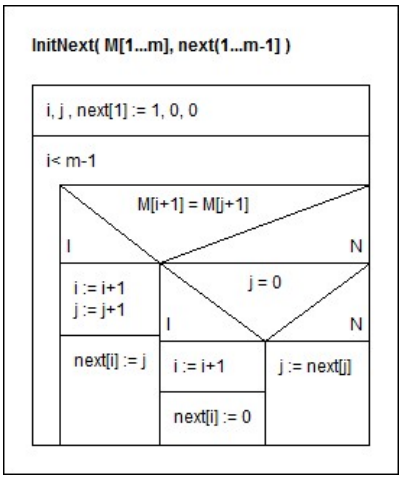


Figure 10: KMP tolásokat szabályzó tömb kitöltése

Az algoritmus (ld 11. ábra):

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

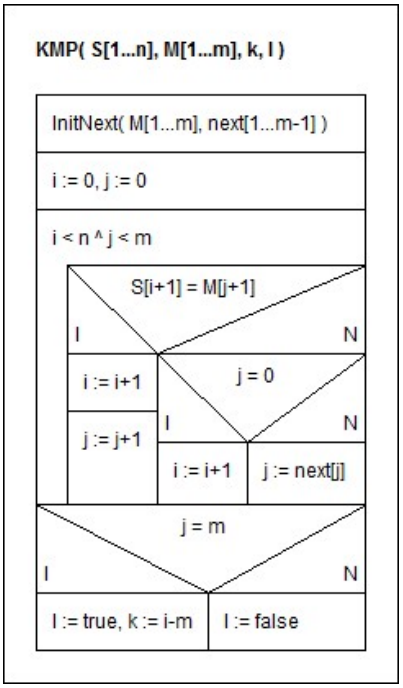


Figure 11: KMP algoritmus

6.3 Boyer-Moore | Quick search algoritmus

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

- Ha a minta utáni karakter benne van a mintában, akkor jobbról az első előfordulására illesztjük. (12. ábra)

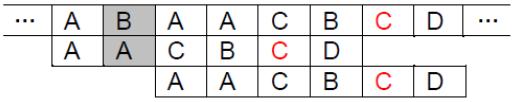


Figure 12: QS - eltolás ha a minta utáni karakter benne van a mintában

- Ha a minta utáni karakter nincs benne a mintában, akkor a mintát ezen karakter után illesztjük. (13. ábra)

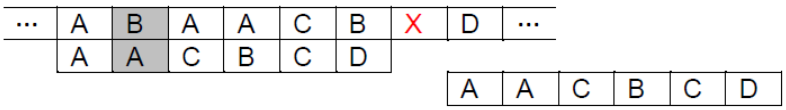


Figure 13: QS - eltolás ha a minta utáni karakter nincs benne a mintában

Az eltolás kiszámítását megint elő lehet segíteni egy tömbbel, most azonban, mivel nem a minta az érdekes, és nem tudjuk pontosan mely karakterek szerepelnek a szövegben, így a tömbbe az egész abc-t fel kell vennünk (14. ábra)

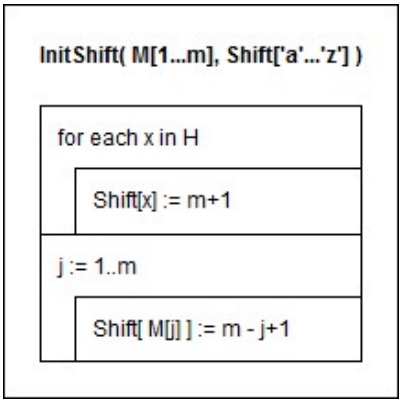


Figure 14: QS - Az eltolást elősegítő tömb ($Shift['a'...'z']$) konstruálása

Az algoritmus (ld. 15. ábra):

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

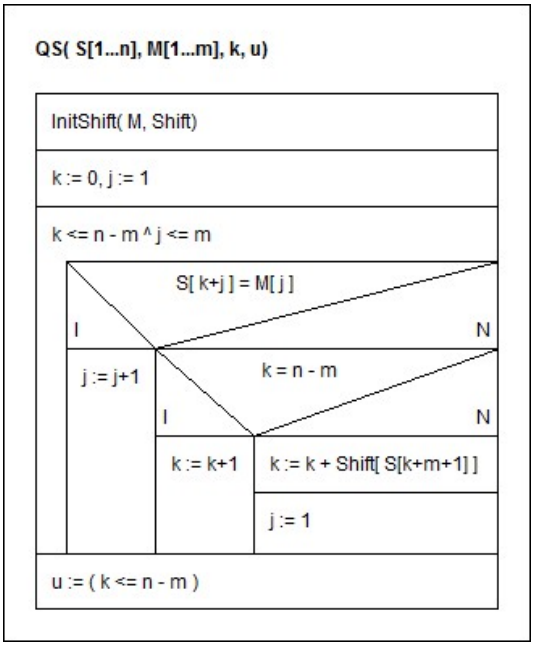


Figure 15: QS

6.4 Rabin-Karp algoritmus

A Rabin-Karp algoritmus lényege, hogy minden betűhöz az ábécéből egy számjegyet rendelünk, és a keresést számok összehasonlításával végezzük. Világos, hogy ehhez egy ábécé méretnek megfelelő számrendszerre lesz szük-

ségünk. A szövegből mindig a minta hosszával egyező részeket szelünk ki, és ezeket hasonlítjuk össze.

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

A fent látható szeletek lesznek az s_i-k.

Az algoritmus működéséhez azonban számos apró ötletet alkalmazunk:

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

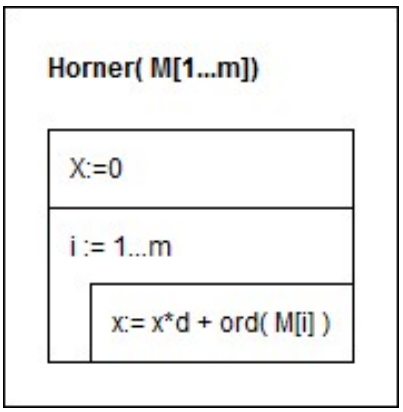


Figure 16: RK - Horner-módszer

   Az *ord*() függvény az egyes betűknek megfelelő számot adja vissza. A d a számrendszer alapszáma.

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

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

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

   Példa:

   Az előző példa szövegével és mintájával ($d = 10$ elemű ábécé és $m = 4$ hosszú minta):
   $s_0 = 3020$, ekkor: $s_{0+1} = s_1 = (3020 - ord(D) \cdot 10^3) \cdot 10 + ord(B) = (3020 - 3000) \cdot 10 + 1 = 0201$

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

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

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

4. A *mod p* számítás egy másik problémát is felvet. Ugyanis a kivonás alkalmával negatív számokat is kaphatunk.

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

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

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

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

A fentiek alapján az algoritmus a következő (ld. 17. ábra)

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

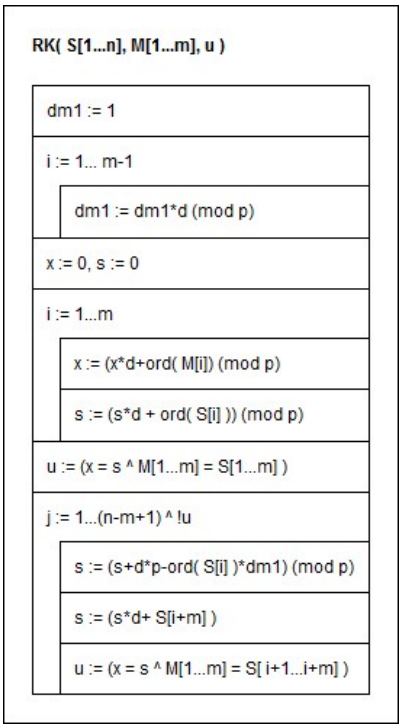


Figure 17: RK