Algoritmusok II. gyakorló feladatok (2016)

Rendezés lineáris időben

1.1. Rendezzük számjegypozíciós listarendezéssel a 013, 200, 010, 321, 213, 201 négyes számrendszerbeli számok egyszerű láncolt listáját! Mutassuk be a fenti példán az algoritmus működését! Tegyük meg ezt a 31, 21, 30, 22, 01, 23 listára is!

1.2. Rendezzük számjegypozíciós (Radix) rendezéssel a 013, 200, 010, 321, 213, 201 négyes számrendszerbeli számok tömbjét! Az egyes menetekben leszámláló rendezést alkalmazzunk! Mutassuk be a fenti példán az algoritmus működését! Tegyük meg ezt a 31, 21, 30, 22, 01, 23 tömbre is!

1.3. Az `A[1..n]` vektort szeretnénk rendezni. A vektor elemtípusának kulcsfüggvénye $\varphi = (\varphi_k, \ldots \varphi_2, \varphi_1)$ alakú, és az elemeket a fenti, összetett kulcsfüggvény szerint lexikografikusan rendezve, monoton növekvően kell sorbarakni.

Adott a `rendezi(A[1..n],i,B[1..n])` eljárás, amely az `A[1..n]` tömböt a φ_i kulcsfüggvény szerint, rendezéstartó módon, monoton növekvően a `B[1..n]` tömbbe rendezi, ahol $T(n,k) \in O(f(n))$. (Ezt nem kell megírni!)

Írjuk meg a `rendez(A[1..n])` eljárást, amely megoldja az eredeti rendezési feladatot, ahol $T(n,k) \in O(k * f(n))$.

1.4. Írjuk meg a `negPoz(A[1..n])` eljárást, ami az `A[1..n]` vektor elejére rendezi a negatív elemeket, a végére pedig a nemnegatívakat! $T(n) \in O(n)$, $M(n) \in O(1)$.

1.5. Adott az `A[1..n]` tömb, amelynek elemei k bites, nemnegatív egész számok. Írjuk meg a `Bincsere(A[1..n])` eljárást, ami a tömböt monoton növekvően, bináris cserével helyben rendezi, ahol $T(n,k) \in O(k * n)$.

Az alkalmazandó algoritmus: A legmagasabb helyiérték szerint kettéosztjuk a tömb elemeit, majd a résztömböket a második legmagasabb helyiérték szerint osztjuk ketté stb.

(**Ötlet:** Használjuk ehhez a `bcs(A[u..v],m)` eljárást, ami az `A[u..v]` résztömböt a legalacsonyabb helyiértékű `m` bit szerint rendezi.)

Hasító táblák

2.1. A `T[0..M-1]` hasító tábla rései (slot-jai) egyirányú, nemciklikus, fejelem nélküli, rendezetlen láncolt listák pointerei. Adott a `hash(kulcs) = kulcs mod M` tördelőfüggvény (hash-függvény). Feltehető, hogy a tábla kulcsok szerint egyértelmű.

2.1.a. Írjuk meg az `ins(T[],M,k,a) : 0..2` értékű függvényt, ami beszúrja a `T[0..M-1]` hasító táblába a `(k,a)` kulcs-adat párt! Akkor és csak akkor ad vissza `0` értéket, ha sikeres volt a beszúrás. Ilyenkor az új listaelemet a megfelelő lista elejére szúrjuk be! Ha a táblában már volt `k` kulcsú elem, a beszúrás meghiúsul, és `2` hibakódot adjunk vissza. Különben, ha nem tudjuk már a szükséges listaelemet allokálni, `1` hibakódot adjunk vissza! Feltehető, hogy a szükséges memória allokátor művelet sikertelenség esetén $\oslash$ pointert ad vissza.

2.1.b. Írjuk meg a `search(T[],M,k)` függvényt, ami visszaadja a T-beli, `k` kulcsú elem címét, vagy a $\oslash$ pointert, ha ilyen nincs!

2.1.c. Írjuk meg a `del(T[],M,k)` logikai függvényt, ami törli a `T[0..M-1]` hasító táblából a `k` kulcsú adatot! Akkor és csak akkor ad vissza `true` értéket, ha sikeres volt a törlés, azaz a táblában volt `k` kulcsú elem. Ilyenkor a feleslegessé váló listaelemet deallokáljuk!

2.1.d Mutassuk be, mi történik, ha sorban beszúrjuk a hasító táblába a következő kulcsokat: 5; 28; 19; 15; 20; 33; 12; 17; 10. Az egyszerűség kedvéért legyen $M = 9$, és legyen a hasító függvény a következő: $h(k) = k \bmod 9$.

2.2. A `T[0..M-1]` hasító táblát nyílt címzéssel ábrázoltuk. Adott a $h_1(kulcs) = kulcs \bmod M$ tördelőfüggvény (hash-függvény). Feltehető, hogy a tábla kulcsok szerint egyértelmű.

2.2.a. Írjuk meg az `ins(T[],M,k,a) : 0..2` értékű függvényt, ami beszúrja a `T[0..M-1]` hasító táblába a `(k,a)` kulcs-adat párt! Akkor és csak akkor ad vissza `0` értéket, ha sikeres volt a beszúrás. Ha a táblában már volt `k` kulcsú elem, a beszúrás meghiúsul, és `2` hibakódot adjunk vissza. Különben, ha a táblában már nincs szabad hely, `1` hibakódot adjunk vissza! A kulcsütközést lineáris próbával oldjuk fel! Ne feledkezzünk el az üres és a törölt rések (slot-ok) megkülönböztetéséről!

2.2.b. Írjuk meg a `search(T[],M,k,a)` logikai függvényt, megkeresi a `T[0..M-1]` hasító táblában a `k` kulcshoz tartozó `a` adatot! Akkor és csak akkor ad vissza `true` értéket, ha sikeres volt a keresés, azaz a táblában volt `k` kulcsú elem. (Sikertelen keresés esetén `a` definiálatlan marad.) A kulcsütközést lineáris próbával oldjuk fel! Ne feledkezzünk el az üres és a törölt rések (slot-ok) megkülönböztetéséről!

2.2.c. Írjuk meg a `del(T[],M,k)` logikai függvényt, ami törli a `T[0..M-1]` hasító táblából a `k` kulcsú adatot! Akkor és csak akkor ad vissza `true` értéket, ha sikeres volt a törlés, azaz a táblában volt `k` kulcsú elem. A kulcsütközést lineáris próbával oldjuk fel! Ne feledkezzünk el az üres és a törölt rések (slot-ok) megkülönböztetéséről!

2.3. Oldjuk meg a 2.2 feladatot négyzetes próba esetére! ($c_1 = c_2 = 1/2$)

2.4. Írjuk meg a 2.2 feladat algoritmusait kettős hash-elés esetére! A másodlagos hash függvény: $h_2(kulcs) = 1 + (kulcs \bmod (M-1))$.

2.5. Mutassuk be a 2.1-2.4 feladatokban szereplő algoritmusok működését papíron, konkrét adatokkal, `M=11` esetén! (Mi lehet a baj a négyzetes próbával?) Vegyünk például egy kezdetben üres hasító táblát, és szúrjuk be egymás után a következő kulcsokat: 10; 22; 31; 4; 15; 28; 17; 88; 59. Ezután töröljük a 17-et, majd próbáljuk megkeresni a 18-at és az 59-et, végül pedig szúrjuk be a 18-at! Nyílt címzés esetén minden egyes művelethez adjuk meg a próbasorozatot is!

Gráfok ábrázolásai

3.1. A `b` pointer egy bináris láncolt ábrázolású általános fa gyökércsúcsára mutat. Csúcsait az `1..n` számok egyértelműen címkézik. Írjuk meg a `b2g(b,Adj[1..n])` eljárást, ami a fát átmásolja az `Adj[1..n]` pointertömb segítségével reprezentált, éllistás ábrázolású gráfba! (A híváskor `Adj[1..n]` nem definiált.) $T(n) \in O(n)$, $M(n) \in O(n)$, ahol n a fa csúcsainak száma.

3.2. Az `Adj[1..n]` pointertömb egy élsúlyozatlan gráf éllistás ábrázolása, amiről tudjuk, hogy gyökeres fa. Írjuk meg a `g2b(Adj[1..n],b)` eljárást, ami a gráfként ábrázolt fát átmásolja a `b` gyökérpointerű, bináris láncolt ábrázolású általános fába! (A híváskor `b` nem definiált hivatkozás.) $T(n) \in O(n)$, $M(n) \in O(n)$, ahol n a fa csúcsainak száma. **Ötlet:** használjunk egy `T[1..n]` pointer segédtömböt, ahol `T[i]` az eljárás által létrehozott fa `i` kulcsú csúcsára mutat! (`i`$\in$`1..n`)

3.3. Egy gráf csúcsai az `1..n` egész számok. A gráf egy bejárásának eredménye a `P[1..n]` vektorban ábrázolt feszítőfa: Minden `i` csúcsra `P[i]` az `i` szülője a fában, kivétel a gyökércsúcs, amire `P[i]==0`, és a bejárás során el nem ért csúcsok, ahol `P[i]==-1`.

Írjuk meg az `fb(P[1..n],t)` eljárást, ami a feszítőfáról binárisan láncolt általános fa másolatot készít! (Az eredmény gyökérpointere `t`.) A láncolt reprezentációban a testvérek listái index szerint növekvően rendezettek legyenek! $T(n) \in O(n)$, $M(n) \in O(n)$

(**Ötlet:** a megoldáshoz használjuk a `T[1..n]` pointer-segédtömböt: `T[i]` mutasson a láncolt fa `i` cimkéjű csúcsára, ha a feszítőfa tartalmazza a csúcsot! A `P[1..n]` vektort fordítva (n-től 1-ig visszafelé) érdemes feldolgozni.)

3.4. Egy gráf csúcsai az `1..n` egész számok. A gráf egy bejárásának eredménye a `P[1..n]` vektorban ábrázolt feszítőfa: Minden `i` csúcsra `P[i]` az

`i` szülője a fában, kivétel a gyökércsúcs, amire `P[i]==0`, és a bejárás során el nem ért csúcsok, ahol `P[i]==-1`.

Erről a `t` binárisan láncolt általános fa másolatot készítettük. A `P[1..n]` vektor már nem érhető el.

Írjuk meg a `bf(t,P[1..n])` eljárást, ami a `t` általános fából rekonstruálja az eredeti `P[1..n]` feszítőfát! $T(n) \in O(n)$, $M(n) \in O(n)$

(**Ötlet:** Töltsük fel a `P[1..n]` vektort `-1` értékekkel, majd járjuk be a `t` általános fát preorder módon úgy, hogy mindig ismerjük az aktuális csúcs szülőjét és ezt be is írjuk a `P[1..n]` vektorba a megfelelő helyre.)

3.5. Egy gráfbejárás eredménye a `P[1..n]` feszítőfa.
`i,j` eleme `1..n` esetén:
`P[i]=j` akkor, ha az `i.` csúcs szülője a feszítőfában a `j.` csúcs.
`P[i]=0` akkor, ha az `i.` csúcs a feszítőfa gyökere.
`P[i]<0` akkor, ha az `i.` csúcs nincs benne a fában.

Írjuk meg a `melysegek(P[1..n],d[1..n])` eljárást, ami kiszámítja a `d[1..n]` vektort, ami sorban az egyes csúcsok feszítőfa-beli mélységeit tartalmazza! A gyökér mélysége `0`. `d[i]` végtelen akkor, ha az `i.` csúcs nincs benne a fában. $T(n) \in O(n)$, $M(n) \in O(n)$. A megoldáshoz *ne* használjunk láncolt adatszerkezetet!

(**Ötlet:** Készítsük el a `melyseg(P[1..n],i,d[1..n])` rekurzív segédeljárást, ami kiszámítja az `i.` csúcs – és ha kell, az ősei – mélységét!

3.6. Egy körmentes irányított gráfot a `G[1..n]` pointertömb segítségével éllistásan ábrázoltunk.

Írjuk meg a `gyf(G[1..n])` függvényt, ami visszaadja a gyökércsúcs indexét, ha a gráf egy gyökeres fa, különben nullát ad vissza! $T(n) \in O(n+e)$, ahol e az élek száma. $M(n) \in O(n)$.

(Ötlet: alkalmazhatunk egy segédtömböt, amibe beírjuk a csúcsok szülőjének indexét.)

Szélességi gráfbejárás

4.1. Szemléltessük a szélességi bejárás működését az alábbi gráfokon az s indexű csúcsból indítva! Mutassuk be a sor, a d és a π értékek alakulását a gyakorlatról ismert módon! Nemdeterminisztikus esetekben mindig a kisebb indexű csúcsot dolgozzuk fel először! Végül rajzoljuk le a mélységi feszítőfát, amit az algoritmus kiszámolt!

4.1.a $s = 5$

	1	2	3	4	5	6
1	0	0	0	1	0	0
2	1	0	1	0	1	0
3	0	0	0	0	0	0
4	0	1	0	0	1	0
5	0	0	1	1	0	0
6	0	0	1	0	1	0

4.1.b[1] $s = 3$

$1 \to 2;\ 4.$ $2 \to 5.$ $3 \to 5;\ 6.$
$4 \to 2.$ $5 \to 4.$ $6.$

4.1.c $s = 5$

$1 \to 4.$ $2 \to 1;\ 3;\ 5.$ $3.$
$4 \to 2;\ 5.$ $5 \to 3;\ 4.$ $6 \to 3;\ 5.$

4.1.d[2], $s = 4$

$1 - 2;\ 5.$ $2 - 1;\ 6.$ $3 - 4;\ 6;\ 7.$ $4 - 3;\ 7;\ 8.$
$5 - 1.$ $6 - 2;\ 3;\ 7.$ $7 - 3;\ 4;\ 6;\ 8.$ $8 - 4;\ 7.$

4.2. Az `Adj[1..n]` pointertömb egy élsúlyozatlan gráf éllistás ábrázolása. Írjuk meg az `SZB(Adj[1..n],s,d[1..n],P[1..n])` eljárást, ami az `s` kezdőcsúcsból indítva szélességi módon járja be a fát! A csúcsok mélységeit a `d[1..n]`, a mélységi feszítőfát pedig a `P[1..n]` tömbben számoljuk ki, amik a híváskor definiálatlan értékeket tartalmaznak! $T(n, e) \in O(n + e)$, $M(n) \in O(n)$, ahol n a gráf csúcsainak, e pedig az éleinek száma.

4.3. Egy erdőben van két mókus, mindkettő ugyanakkorát tud ugrani. Két különböző fán vannak. Mindkettő elkezd fáról fára ugrálni. Ugyanannyit ugranak, de nem okvetlenül egyszerre. Kérdés, közben találkozhatnak-e?

Modellezzük a problémát, és írjunk rá struktogramot!

$T(n, e) \in O(n + e)$, ahol n a fák, e pedig az erdőben a mókusok számára lehetséges, fáról fára való ugrások száma. (Vigyázat, nem biztos, hogy egy ilyen ugráshoz mindig lehetséges a visszaugrás is! Másrészt az e szám nem a tényleges ugrások száma, hanem a lehetséges, egymástól páronként különböző, fáról fára való ugrások száma.)

[1] $u \to v_1; \ldots v_n.$ azt jelenti, hogy a gráfnak a következő irányított élei vannak: $(u, v_1), \ldots (u, v_n)$.

[2] $u - v_1; \ldots v_n.$ azt jelenti, hogy a gráfnak a következő irányítatlan élei vannak: $(u, v_1), \ldots (u, v_n)$.

Ötlet: Indítsunk két m mélységű szélességi gráfbejárást a két adott csúcsból (az m mélységű csúcsok gyerekeit már nem tesszük a sorba). Akkor találkozhatnak, ha a második bejárás elér legalább egy, az első által is érintett csúcsot.

Mélységi gráfbejárás

5.1.a Szemléltessük a mélységi bejárás működését az alábbi gráfon! Írjuk az elérési számokat és a befejezési számokat a gráf grafikus ábrázolásán a csúcsok mellé! Jelezzük a különböző éltípusokat a szokásos módon! Nemdeterminisztikus esetekben mindig a kisebb indexű csúcsot dolgozzuk fel először! Adjuk meg a gráf csúcsainak a mélységi bejárásból adódó topologikus rendezését!

$1 \to 2.$ $\quad 2 \to 3;\ 5.$ $\quad 3.$
$4 \to 2;\ 5.$ $\quad 5 \to 3;\ 6.$ $\quad 6.$

5.2. Egy irányított gráf csúcsait az `1..n` számokkal címkéztük. A gráfot a `C[1..n,1..n]` csúcsmátrix segítségével ábrázoljuk. Feltesszük, hogy a gráf nem tartalmaz irányított kört. Csúcsai egy topologikus rendezését, azaz ütemezését állítjuk elő a `TR[1..n]` vektorban. Adott a `be[1..n]` tömb, ami kezdetben a csúcsok bemeneti fokszámait tartalmazza. Egy `i` csúcs akkor ütemezhető be, ha `be[i]==0`. Ha egy csúcsot beütemezünk, akkor a közvetlen rákövetkezői `be[j]` értékeit eggyel csökkentjük.
(Struktogram: `TopRend(C[1..n,1..n],be[1..n],TR[1..n])`)
$T(n) \in O(n^2)$, $M(n) \in O(n)$

5.3. Írjunk struktogramot gráfok a topologikus rendezésének alábbi változatára!
Egy irányított gráf csúcsait az `1..n` számokkal címkéztük. A gráfot éllistásan ábrázoltuk a `G[1..n]` pointertömb segítségével. Ha a gráf nem tartalmaz irányított kört, csúcsai egy topologikus rendezését, azaz ütemezését állítjuk elő a `TR[1..n]` vektorban, és true értékkel térünk vissza. (Ha egy csúcsból egy másikba irányított út vezet, akkor az első a parciális rendezésben kisebb sorszámmal szerepel. Ha két csúcs között nincs irányított út, a parciális rendezésben a sorrendjük közömbös.) Ha a gráf tartalmaz irányított kört, false értékkel térünk vissza. A struktogramot az alábbi algoritmus szerint kell megírni.
(Struktogram: `TopRend(G[1..n],be[1..n],TR[1..n]):bool`)
$T(n,e) \in O(n+e)$, ahol e az élek száma; $M(n) \in O(n)$

Algoritmus:

Adott a `be[1..n]` tömb, ami kezdetben a csúcsok bemeneti fokszámait tartalmazza. (Ez tehát a programunk számára adott, nem kell megírni.) Egy

`i` csúcs akkor ütemezhető be, ha még nem ütemeztük be, és `be[i]=0`. Az algoritmus minden lépésében egy beütemezhető csúcsot ütemezünk be, azaz betesszük a `PR[1..n]` vektorba, „az első szabad helyre". A beütemezhető csúcsok egy `Q` sorban vannak. Ha egy csúcsot beütemezünk, akkor a közvetlen rákövetkezői `be[j]` értékeit eggyel csökkentjük. Ha a sor elfogy, aszerint térünk vissza true értékkel, hogy minden csúcsot beütemeztünk-e.

5.4. Egy irányított gráfot a `G[1..n]` pointertömb segítségével éllistásan ábrázoltunk.

Írjuk meg a `gye(G[1..n],Q)` eljárást, ami visszaadja a `Q` sorban a gyökércsúcsok indexeit, ha a gráf egy erdő, azaz gyökeres fák halmaza, különben a `Q` sort üresen adja vissza! $T(n) \in O(n+e)$, ahol e az élek száma. $M(n) \in O(n)$.

Ötlet: Írjunk mélységi bejárást, a következő módosítással. Ha fehér csúcsot találunk, a szokásos módon folytatjuk. Ha szürke csúcsot találunk, akkor irányított kört is találtunk, és a gráf nem erdő. Ha fekete csúcsot találunk, két eset van. (1) Ha ez (a mondjuk `j` csúcs) egy korábban felépített részleges feszítőfa gyökere (`P[j]=\NIL`), akkor becsatoljuk abba a fába, amit éppen most építünk a `P[1..n]` tömbben. (2) Ha nem gyökércsúcsot találtunk, akkor ennek a csúcsnak a bemeneti fokszáma nagyobb mint `1`, tehát a gráf nem erdő. Ha végigmegy a bejárás, és nem találunk sem szürke, sem második típusú fekete csúcsot, akkor a gráf egy erdő, csak a fái gyökércsúcsait a `P[1..n]` tömbből a `Q` sorba kell másolni. Különben `Q` üresen marad.

5.5. Az alábbi gráfon szemléltessük a gyakorlatról ismert módon
(a) a mélységi bejárásos
(b) a csúcsok bemeneti fokszámait felhasználó
topologikus rendezés algoritmusát! (Ez két külön feladat.)

$1 \to 2.$ $2 \to 3;\ 5.$ $3 \to 6.$
$4 \to 5.$ $5 \to 6.$ $6.$

Minimális feszítőfák

6.1. Az alábbi egyszerű gráfon szemléltessük a gyakorlatról ismert módon (a) a Kruskal algoritmust, (b) az egyes sorszámú csúcsból indított Prim algoritmust! Mindkét esetben adjuk meg a kapott minimális feszítőfát is! (A „-" jelek a „végtelen" szimbólumot helyettesítik.)

	1	2	3	4	5	6
1	0	4	-	1	-	-
2	4	0	1	2	0	-
3	-	1	0	-	1	1
4	1	2	-	0	0	-
5	-	0	1	0	0	1
6	-	-	1	-	1	0

6.2.* Általánosítsuk a Prim algoritmust arra az esetre, ha a gráf nem összefüggő, és így nem tudunk feszítőfát, csak feszítő erdőt megadni! A gráfot a szimmetrikus `C` költségmátrix segítségével ábrázoljuk. (Struktogram: `Prim(C[1..n,1..n],d[1..n],P[1..n])`)
$T(n) \in O(n^2)$, $M(n) \in O(n)$

6.3.* Írjunk struktogramot a Prim algoritmus alábbi változatára!
Egy nem okvetlenül összefüggő, súlyozott irányítatlan gráf csúcsait az `1..n` számokkal címkéztük. A gráfot éllistásan ábrázoltuk a `G[1..n]` pointertömb segítségével. A gráf egy minimális feszítő erdőjét állítjuk elő a Prim algoritmus segítségével, a `P[1..n]` vektorban. A minimális feszítőerdő fáit a gráf összefüggő komopenseinek minimális feszítő fái adják. A feszítőfákat a `P[1..n]` vektorban irányított fákként ábrázoljuk, és minden él az őt tartalmazó feszítőfa gyökere felé irányítva jelenik meg. A `d[1..n]` vektor a `P[1..n]` vektorban visszadott élek súlyait fogja tartalmazni: `j>0` esetén `P[i]=j` jelentése, hogy az `(i,j)` él eleme az egyik feszítőfának. Ilyenkor `d[i]` tartalmazza az `(i,j)` él súlyát (azaz költségét). `P[i]=0` jelentése, hogy `i` az egyik feszítőfa gyökere. Ilyenkor `d[i]=0` lesz. Ha a Prim algorimus során egy csúcs még nincs feszítőfában, de szomszédja az éppen épített feszítőfa egy vagy több csúcsának, akkor azt mondjuk, hogy szomszédja ennek a feszítőfának, és a feszítőfától való távolsága egy minimális súlyú él költsége, ami őt a feszítőfához kapcsolja. A struktogramot a fenti fogalmak és ábrázolás, valamint az alábbi algoritmus szerint kell megírni. (Struktogram: `Prim(G[1..n],P[1..n],d[1..n])`) $T(n) \in O(n^2)$, $M(n) \in O(n)$

Algoritmus:
I. Kezdetben egyik csúcs sincs feszítőfában.
`P[1..n]:=-1; d[1..n]:=INFINITE;`
II. Válasszunk ki egy tetszőleges `s` csúcsot, ami még nincs feszítőfában! Legyen ez egy új feszítőfa gyökere! `P[s]:=0; d[s]:=0`
III. A most kiválasztott csúcs minden olyan `j` szomszédjára, ami még nincs a feszítőfában, állítsuk be a `P[j]` és `d[j]` értékeket, a `j` csúcsnak a feszítőfától való távolsága szerint!

IV. Ha a feszítőfának van szomszédja, válasszunk ki egyet, ami minimális távolságra van tőle (ezt a `d[1..n]` tömbön egy feltételes minimumkiválasztással oldjuk meg), és a kiválasztott csúcsot vegyük hozzá a feszítőfához, majd folytassuk a III. ponttól!
V. Ha a feszítőfának nincs szomszédja, akkor az aktuális feszítőfa kész van.
VI. Ha van még olyan csúcs ami egyetlen feszítőfában sincs benne, folytassuk a II. ponttól!
VII. Különben a feszítőerdő is kész van.

6.4. Adjuk meg Kruskal algoritmusában az Unió-HolVan adatszerkezetet inicializáló eljárás, továbbá a HolVan függvény és az Unió eljárás struktogramját!

Legrövidebb utak egy forrásból: Sor alapú Bellman-Ford algoritmus

7.1. Írjuk meg Sor alapú Bellman-Ford algoritmust arra az esetre, ha a gráfot éllistásan ábrázoljuk a `G[1..n]` pointertömb segítségével! (Struktogram: QBF($G[1..n], d[1..n], P[1..n]$) eljárás.) $T(n) \in O(n * e)$, ahol e az élek száma. $M(n) \in O(1)$.

7.2.** Próbáljuk meg a fenti Sor alapú Bellman-Ford algoritmust kiegészíteni a negatív kör figyelésével! Alakítsuk át az eljárást függvénnyé, ami 0 értéket ad vissza, ha nem talált negatív kört, különben pedig a negatív kör egy csúcsának indexét. Ez utóbbi esetben `d[1..n]` és `P[1..n]` definiálatlan marad.

7.3. Szemléltessük a Sor alapú Bellman-Ford FIFO algoritmus működését az alábbi gráfon a `4` indexű csúcsból indítva! Mutassuk be a `d[1..6]` (elérési költségek) és a `P[1..6]` (feszítőfa) tömbök, valamint a sor alakulását! Egy sor az egy csúcsból kimenő élek feldolgozása legyen! Nemdeterminisztikus esetekben mindig a kisebb indexű csúcsot dolgozzuk fel először!

1 → 2, 2; 5, 5.	2 → 1, 2; 5, 1.	3 → 2, 1.
4 → 1, 1; 2, 4; 3, 1.	5.	6.

Legrövidebb utak egy forrásból körmentes irányított gráfokra

8.1. Szemléltessük az alábbi gráfon a DAG legrövidebb utak egy forrásból algoritmus működését, ha a kettes csúcs a startcsúcs!

1 → 2, 2; 5, 5.	2 → 3, 2; 5, 3.	3 → 4, -1; 5, 1.
4 → 5, 1; 6, 4.	5. → 6, 2	6.

Legrövidebb utak egy forrásból: Dijkstra algoritmus

9.1. Szemléltessük a Dijkstra algoritmus működését az alábbi gráfon, a `3` indexű csúcsból indítva! (A „-” jel a „plusz végtelen”-t jelöli.) Mutassuk be a `d[1..6]` (elérési költségek) és a `P[1..6]` (feszítőfa) tömbök alakulását! Minden sor mellett jelezzük, hogy az melyik csúcs kiterjesztéséből adódott! Nemdeterminisztikus esetekben mindig a kisebb indexű csúcsot dolgozzuk fel először! Adjuk meg a kapott feszítőfát is!

	1	2	3	4	5	6
1	0	1	-	1	-	-
2	-	0	-	5	2	-
3	-	1	0	-	-	1
4	-	-	-	0	-	-
5	-	2	-	1	0	-
6	-	-	-	-	1	0

9.2. Az alábbi egyszerű gráfon szemléltessük a gyakorlatról ismert módon az egyes sorszámú csúcsból indított Dijkstra algoritmust! Adjuk meg a kapott feszítőfát is! (A „-” jelek a „végtelen” szimbólumot helyettesítik.)

	1	2	3	4	5	6
1	0	4	-	1	-	-
2	4	0	1	2	0	-
3	-	1	0	-	1	1
4	1	2	-	0	0	-
5	-	0	1	0	0	1
6	-	-	1	-	1	0

Floyd-Warshall algoritmus

10.1. Szemléltessük a Floyd-Warshall algoritmus működését az alábbi gráfon a $(D^{(0)}, \Pi^{(0)}), \ldots, (D^{(3)}, \Pi^{(3)})$ mátrix párok megadásával! A gráf az alábbi csúcsmátrixszal adott.

	1	2	3
1	0	5	3
2	1	0	-
3	3	1	0

10.2. Szemléltessük a Floyd-Warshall algoritmus működését az alábbi gráfon a $(D^{(0)}, \Pi^{(0)}), \ldots, (D^{(4)}, \Pi^{(4)})$ mátrix párok megadásával! A gráf az alábbi csúcsmátrixszal adott.

	1	2	3	4
1	0	5	3	1
2	5	0	1	-
3	3	1	0	1
4	1	-	1	0

Gráfok tranzitív lezártja

11.1. Szemléltessük Warshall, gráfok tranzitív lezártját meghatározó algoritmusának működését az alábbi (csúcsmátrixszal adott) gráfon a $T^{(0)}, \ldots, T^{(3)}$ mátrixok megadásával!

	1	2	3
1	0	0	1
2	1	0	0
3	0	1	0

11.2. Szemléltessük Warshall, gráfok tranzitív lezártját meghatározó algoritmusának működését az alábbi (csúcsmátrixszal adott) gráfon a $T^{(0)}, \ldots, T^{(4)}$ mátrixok megadásával!

	1	2	3	4
1	0	1	0	0
2	0	0	1	0
3	0	0	0	1
4	1	0	0	0

32 String Matching

32.1 The naive string-matching algorithm

32.1-1 Show the comparisons the naive string matcher makes for the pattern $P = 0001$ in the text $T = 000010001010001$.

32.1-2 Suppose that all characters in the pattern P are different. Show how to accelerate NAIVE-STRING-MATCHER to run in time $O(n)$ on an n-character text T .

32.2 The Rabin-Karp algorithm

32.2-1 Working modulo $q = 11$, how many spurious (i.e. false) hits does the Rabin-Karp matcher encounter in the text $T = 3141592653589793$ when looking for the pattern $P = 26$?

32.2-2 How would you extend the Rabin-Karp method to the problem of searching a text string for an occurrence of any one of a given set of k patterns? Start by assuming that all k patterns have the same length. Then generalize your solution to allow the patterns to have different lengths.

32.2-3 Show how to extend the Rabin-Karp method to handle the problem of looking for a given $m \times m$ pattern in an $n \times n$ array of characters ($1 \leq m \leq n$). (The pattern may be shifted vertically and horizontally, but it may not be rotated.)

32.4 The Knuth-Morris-Pratt algorithm

32.4-1 Compute $next[1..19]$ (also called $prefix$ or π function) for the pattern *ababbabbabbababbabb*.

32.4.2a Compute $next[1..4]$ for the pattern $P = 0001$. Show the comparisons the Knuth-Morris-Pratt string matcher makes for this pattern in the text $T = 000010001010001$.

32.4.2b Compute $next[1..5]$ for the pattern $P = abaab$. Show the comparisons the Knuth-Morris-Pratt string matcher makes for this pattern in the text $T = aaababaabaababaab$.

32.4.2c Compute $next[1..6]$ for the pattern $P = aabaab$. Show the comparisons the Knuth-Morris-Pratt string matcher makes for this pattern in the text $T = aaabaabaabaababaab$.

32.4.2d Compute $next[1..6]$ for the pattern $P = babbab$. Show the comparisons the Knuth-Morris-Pratt string matcher makes for this pattern in the text $T = ababbabbababbababbabb$.

32.4.2e Compute $next[1..7]$ for the pattern $P = ABABAKI$.
Show the comparisons the Knuth-Morris-Pratt string matcher makes for this pattern in the text
$T = BABALABABATIBABABAKI$.

32.4-3 Explain how to determine the occurrences of pattern P in the text T by examining $next[1..|PT|]$.

32.4-7 Give a linear-time algorithm to determine whether a text T is a cyclic rotation of another string T'. For example, *arc*, *rca*, and *car* are cyclic rotations of each other.

32.x The Quick Search algorithm

<http://aszt.inf.elte.hu/~asvanyi/ds/Quick Searching.ppt>

32.x.1 Compute $shift[0..1]$ for the pattern $P = 000$. Show the comparisons the Quick Search string matcher makes for this pattern in the text $T = 000010001010001$.

32.x.2 Compute $shift['A'..'F']$ for the pattern
$P = ABABACD$. Show the comparisons the Quick Search string matcher makes for this pattern in the text
$T = BABAEABABAFDBABABACD$.

32.x.3 Compute $shift['A','C','G','T']$ for the pattern
$P = GCAGAGAG$. Show the comparisons the Quick Search string matcher makes for this pattern in the text
$T = GCATCGCAGAGAGTATACAGTACG$.

DC Data Compression

DC.1 The Huffman coding

<http://en.wikipedia.org/wiki/Huffman_coding>

DC.1.1 We would like to compress the text

MATEKFELELETEMKETTESLETT

with Huffman coding. Draw the Huffman tree, give the Huffman code of each letter, the Huffman code of the text, and the length of the latest in bits. Show the letters of the original text in the Huffman code of it.

DC.1.2 Solve Exercise DC.1.1 with the following text.

EMESE MAI SMSE NEM NAIV MESE

DC.2 The Lempel–Ziv–Welch (LZW) algorithm

<http://en.wikipedia.org/wiki/Lempel-Ziv-Welch>

DC.2.1 We have compressed a text with the Lempel-Ziv-Welch algorithm. The text contains letters 'A', 'B', and 'C'. Their codes are 1, 2, and 3 in turn. While building the dictionary, for the code of each new word the first unused positive integer was selected. In this way we have received the following code.

1 2 4 3 5 6 9 7 1

Give the original text, and the complete dictionary.

DC.2.2 Solve Exercise DC.2.1 with the following LZW code.

1 2 4 3 5 8 1 10 11 1