Záróvizsga Tételek 2022

Algoritmusok és Adatszerkezetek I

1. Részproblémára bontható algoritmusok (mohó, oszd-meg-és-uralkodj, dinamikus programozás), rendező algoritmusok, gráfalgoritmusok (szélességi- és mélységi keresés, minimális feszítőfák, legrövidebb utak)

Mohó algoritmusok

A feladatot pontosan egy részfeladatra bontják, és azt tovább rekurzívan oldják meg. Mindig a legjobbnak tűnő megoldás irányába haladunk tovább.

Nem minden problémára adható mohó megoldás!

De ha létezik, akkor nagyon hatékony!

Mohó választás: Az adott problémát egyetlen részproblémára bontja. Ennek optimális megoldásából következik az eredeti feladat optimális megoldása is.

Mohó algoritmus tervezése

1. Fogalmazzuk meg a **mohó választást**.
2. Bizonyítsuk be, hogy az eredeti problémának mindig van olyan **optimális megoldása**, amely **tartalmazza a mohó választást**. Tehát hogy a mohó választás **biztonságos**.
3. Bizonyítsuk be, hogy a mohó választással olyan részprobléma keletkezik, amelynek egy **optimális megoldásához hozzávéve a mohó választást**, az eredeti probléma egy optimális megoldását kapjuk.

Példa: *Töredékes hátizsák feladat*

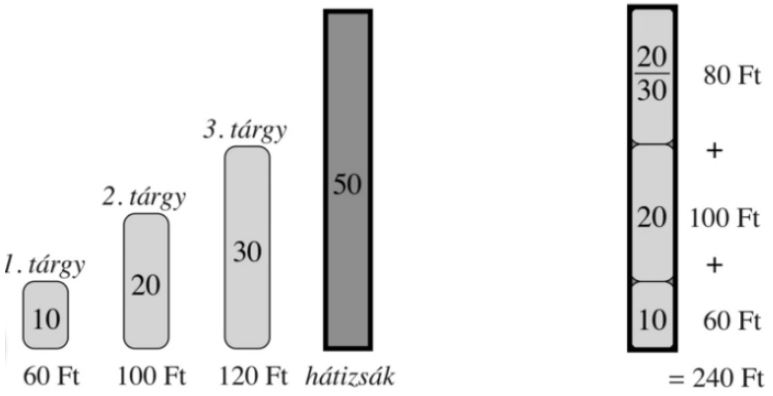
Bemenet: A hátizsák S kapacitása, n tárgy, S_i tárgy súlyok, E_i tárgy értékek

Kimenet: Mi a legnagyobb érték, ami S kapacitásba belefér?

Minden tárgyból 1db van, de az darabolható.

Algoritmus:

- Számoljuk ki minden tárgyra az $\frac{E_i}{S_i}$ arányt
- Tegyünk bele a legnagyobb $\frac{E_i}{S_i}$-vel rendelkező, még rendelkezésre álló tárgyból annyit a zsákba, amennyi belefér



Futás a fenti példán:

- Kiszámoljuk az $\frac{E_i}{S_i}$ értékeket
    i. Tárgy: 6
    ii. Tárgy: 5
    iii. Tárgy: 4
- Végighaladunk a tárgyakon az $\frac{E_i}{S_i}$ arányok szerint
    - Az első tárgy teljes egészében belefér, azt beválasztjuk.
    - A 2. tárgy is teljes egészében belefér, azt is beválasztjuk.
    - A 3. tárgy már nem fér be, beválasztunk annyit, amennyi kitölti a szabad helyet. Jelen esetben a tárgy $\frac{2}{3}$-át.

> A probléma nem-törtedékes verziójára ez a mohó algoritmus nem mindig talál optimális megoldást.

> test

Oszd-meg-és-uralkodj algoritmusok

A feladatot több **részfeladatra** bontjuk, ezek hasonlóak az eredeti feladathoz, de méretük kisebb, tehát ugyan azt a feladatot akarjuk egy kisebb bemenetre megoldani.

Rekurzív módon megoldjuk ezeket a részfeladatokat (azaz ezeket is kisebb részfeladatokra bontjuk egészen addig, amíg elemi feladatokig jutunk, amelyekre a megoldás triviális), majd **összevonjuk őket**, hogy az eredeti feladatra megoldást adjanak.

> A részfeladatok ne legyenek átfedőek. Bár az algoritmus ettől még működhet, de nem hatékony.

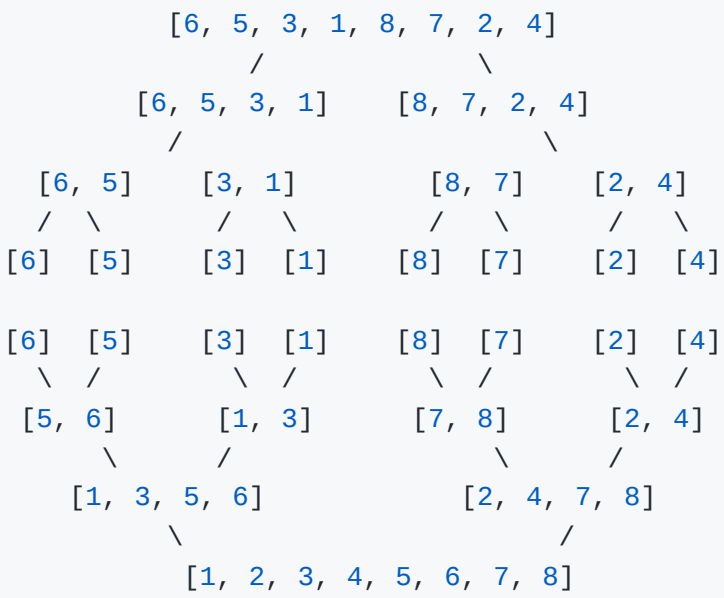
Lépések

1. **Felosztás**: Hogyan osztjuk fel a feladatot több kisebb részfeladatra.
2. **Uralkodás**: A feladatokat rekurzív módon megoldjuk. Ha a részfeladatok mérete elég kicsi, akkor közvetlenül meg tudjuk oldani a részfeladatot, ilyenkor nem osztjuk tovább rekurzívan.

3. **Összevonás**: A részfeladatok megoldásait összevonjuk az eredeti feladat megoldásává.

Példa: *Összefésülő rendezés*

1. **Felosztás**: Az n elemű rendezendő sorozatot felosztja két $\frac{n}{2}$ elemű részsorozatra.
2. **Uralkodás**: A két részsorozatra rekurzívan tovább hívjuk az összefésülő rendezés eljárást. Az elemi eset az egy elemű részsorozat, hiszen az már rendezett, ilyenkor csak visszatérünk vele.
3. **Összevonás**: Összefésüli a két rendezett részsorozatot, ezzel létrehozza az eredeti sorozat rendezett változatát.



> Az összefésülés folyamata egyszerű, csak két mutatót vezetünk a két rendezett tömbön, lépkedünk, mindig a kisebbet fűzzük egy másik, kezdetben üres tömbhöz.

Példa: *Felező csúcskereső algoritmus*

Vizsgáljuk meg a középső elemet. Ha csúcs, térjünk vissza vele, ha nem csúcs, akkor az egyik szomszédja nagyobb, vizsgáljuk tovább a bemenet felét ezen szomszéd irányába. Azért megyünk ebbe az irányba, mert erre biztosan van csúcs. Ezt onnan tudjuk, hogy maga ez a nagyobbik szomszéd is egy potenciális csúcs. Ha mindkét szomszédja nagyobb, akkor mindegy melyik irányba haladunk tovább, egyszerűen azzal, amiről előbb megtudtuk, hogy nagyobb.

1. **Felosztás**: n elemű sorozatot felosztjuk két $\frac{n-1}{2}$ elemű részsorozatra
2. **Uralkodás**: A megfelelő részsorozatban rekurzívan tovább keresünk csúcsot
3. **Összevonás**: Ha csúcsot találtunk, adjuk vissza

```
// Kiindulási tömb:
[1, 3, 4, 3, 5, 1, 3]

// Középső elemet megkeressük, nem csúcs, így tovább haladunk:
[1, 3, 4, 3, 5, 1, 3]
          ^
```

```
// Középső elemet megkeressük, nem csúcs, így tovább haladunk:
[1, 3, 4][3, 5, 1, 3]
    ^
// A középső elem egy csúcs, visszaadjuk
[1, 3][4][3, 5, 1, 3]
       ^
```

Ez az algoritmus logaritmikus időigényű. Ezzel szemben az egyszerű megoldás amikor minden elemen végighaladva keresünk csúcsot, lineáris, azaz jelentősen rosszabb.

Dinamikus programozás

Olyan feladatok esetén alkalmazzuk, amikor a **részproblémák nem függetlenek**, azaz vannak közös részproblémák.

> **Optimalizálási feladatok** tipikusan ilyenek.

A megoldott **részproblémák eredményét memorizáljuk** (mondjuk egy táblázatban), így ha azok mégegyszer elő kerülnek, nem kell újra kiszámolni, csak elővenni memóriából az eredményt.

Iteratív megvalósítás

- Minden részmegoldást kiszámolunk.
- Alulról-felfelé építkező megközelítés, hiszen előbb a kisebb részproblémákat oldjuk meg, amiknek az eredményét felhasználjuk az egyre nagyobb részproblémák megoldásához.

Rekurzív megvalósítás

- Részmegoldásokat kulcs-érték formájában tároljuk.
- Felülről lefele építkező megközelítés.
- **Csak akkor használjuk, ha nem kell minden megoldást kiszámolni!**
    - Ha ki kell mindent számolni, érdemesebb az iteratív megköelítést választani a függvényhívások overhead-je miatt.

Példa: *Pénzváltás feladat*

Adott P_i érmékkel (mindből van végtelen sok) hogyan lehet a legkevesebb érmét felhasználva kifizetni F forint.

```
// Input:
P1 = 1;
P2 = 5;
P3 = 6;
F  = 9;
```

Rekurzív megvalósítással a futás

```
// Egy dimenziós tömbbel dolgozunk, egyes sorokban
// az egyes hívások állapota látszódik.
// Első sor a pénzérme indexét jelöli.

0  1  2  3  4  5  6  7  8  9
0  -  -  -  -  -  -  -  -  ? // penzvalt(9) = min( penzvalt(3), penzvalt(4), penzva
0  -  -  ?  -  -  -  -  -  ? // penzvalt(3) = min( penzvalt(2) ) + 1
0  -  ?  ?  -  -  -  -  -  ? // penzvalt(2) = min( penzvalt(1) ) + 1
0  ?  ?  ?  -  -  -  -  -  ? // penzvalt(1) = min( penzvalt(0) ) + 1
0  1  ?  ?  -  -  -  -  -  ? // penzvalt(0)-t ismertük már, kiindulástól kezdődően
0  1  2  ?  -  -  -  -  -  ? // penzvalt(1) visszatér, kiadja penzvalt(2) eredmény
0  1  2  3  -  -  -  -  -  ? // penzvalt(2) visszatér, kiadja penzvalt(3) eredmény
0  1  2  3  -  -  -  -  -  ? // penzvalt(3) visszatér

// penzvalt(9) jelenleg itt tart: min( 3, penzvalt(4), penzvalt(8) ) + 1
0  1  2  3  4  -  -  -  -  ? // penzvalt(4) = min( penzvalt(3) ) + 1

// penzvalt(9) jelenleg itt tart: min( 3, 4, penzvalt(8) ) + 1
0  1  2  3  4  -  -  -  ?  ? // penzvalt(8) = min( penzvalt(2) = 2, penzvaltas(3) =
0  1  2  3  4  -  -  ?  ?  ? // penzvalt(7) = min( penzvalt(1) = 1, penzvaltas(2) =
0  1  2  3  4  -  ?  ?  ?  ? // penzvalt(6) -> mivel ilyen érménk van, így ezt nem
0  1  2  3  4  -  1  2  ?  ? // penzvalt(6) visszatér, kiadja penzvalt(7)-et
0  1  2  3  4  -  1  2  3  ? // penzvalt(7) visszatér, kiadja penzvalt(8)-at
0  1  2  3  4  -  1  2  3  4 // penzvalt(8) visszatér, kiadja penzvalt(9)-et
```

> Bár elmondható, hogy egy esetre, az 5-re nem kellett kiszámolnunk az értéket, de ez implementáció függő volt, ha `penzvalt(6)`-ot is ugyan úgy számoltuk volna, mint a többi értéket, akkor mindent kiszámoltunk volna, és a rekurzív függvényhívősok overhead-je miatt egyértelműen az iteratív megközelítés lenne a jobb.

Iteratív megvalósítással a futás

```
// 0-tól F-ig (9-ig) építunk egy egy dimentziós tömböt
0  1  2  3  4  5  6  7  8  9

0  ?  ?  ?  ?  ?  ?  ?  ?  ?
0  1  ?  ?  ?  ?  ?  ?  ?  ? // penzvalt[1] = min( penzvalt[0] ) + 1
0  1  2  ?  ?  ?  ?  ?  ?  ? // penzvalt[2] = min( penzvalt[1] ) + 1
0  1  2  3  ?  ?  ?  ?  ?  ? // penzvalt[3] = min( penzvalt[2] ) + 1
0  1  2  3  4  ?  ?  ?  ?  ? // penzvalt[4] = min( penzvalt[3] ) + 1
0  1  2  3  4  1  ?  ?  ?  ? // penzvalt[5] = min( penzvalt[0], penzvalt[4] ) + 1
0  1  2  3  4  1  1  ?  ?  ? // penzvalt[6] = min( penzvalt[0], penzvalt[1], penzva
0  1  2  3  4  1  1  2  ?  ? // penzvalt[7] = min( penzvalt[1], penzvalt[2], penzva
0  1  2  3  4  1  1  2  3  ? // penzvalt[8] = min( penzvalt[2], penzvalt[3], penzva
0  1  2  3  4  1  1  2  3  4 // penzvalt[9] = min( penzvalt[3], penzvalt[4], penzva
```

Rendező algoritmusok

Rendezés

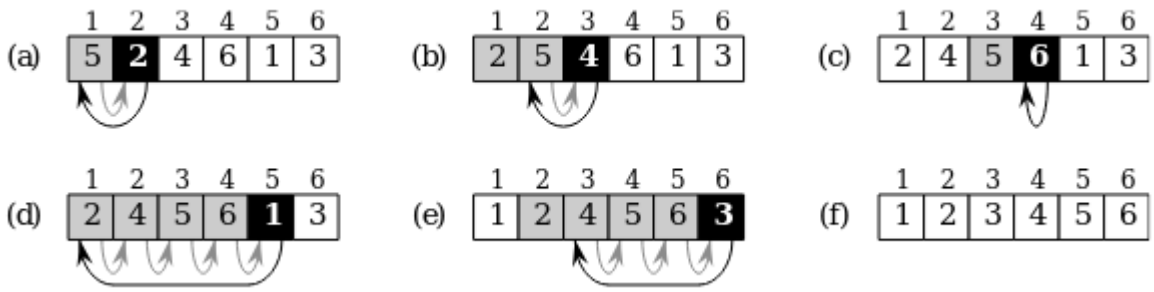
- **Input**: Egészek egy n hosszú tömbje (egy `<a1, a2, ..., an>` sorozat)
- **Output**: n hosszú, rendezett tömb (az input sorozat egy olyan `<a'1, a'2, ..., a'n>` permutációja, ahol `a'1 <= a'2 <= ... <= a'n`)

> Ez egy egyszerű eset, a gyakorlatban:
>
> - Van valamilyen iterálható kollekciónk: `Iterálható<Objektum>`)
> - Van egy függvényünk, ami megondja képt kollekció-elemről, hogy melyik a *nagyobb*: `(a: Objektum, b: Objektum) => -1 | 0 | 1`
>
> Ezek együttesével már megfelelően absztrakt módon tudjuk használni az összehasonlító rendező algoritmusokat bármilyen esetben.

Beszúró rendezés

Helyben rendező módszer.

```
const beszuroRendezes = (A: number[]) => {
  for (let j = 1; j < A.length; j++) {
    const beillesztendo = A[j];
    let i = j - 1;
    for (; i >= 0 && A[i] > beillesztendo; i--) {
      A[i + 1] = A[i];
    }
    A[i + 1] = beillesztendo;
  }
  return A;
};
```



Végig haladunk a tömbön, és minden elemtől visszafelé elindulva megkeressük annak a helyét, és beszúrjuk oda. Amin áthaladtunk, az a részsorozat már rendezett lesz mindig.

Futásidő	Tárigény (össz ~ inputon kívül)
$O(n^2)$	$O(n) \sim O(1)$

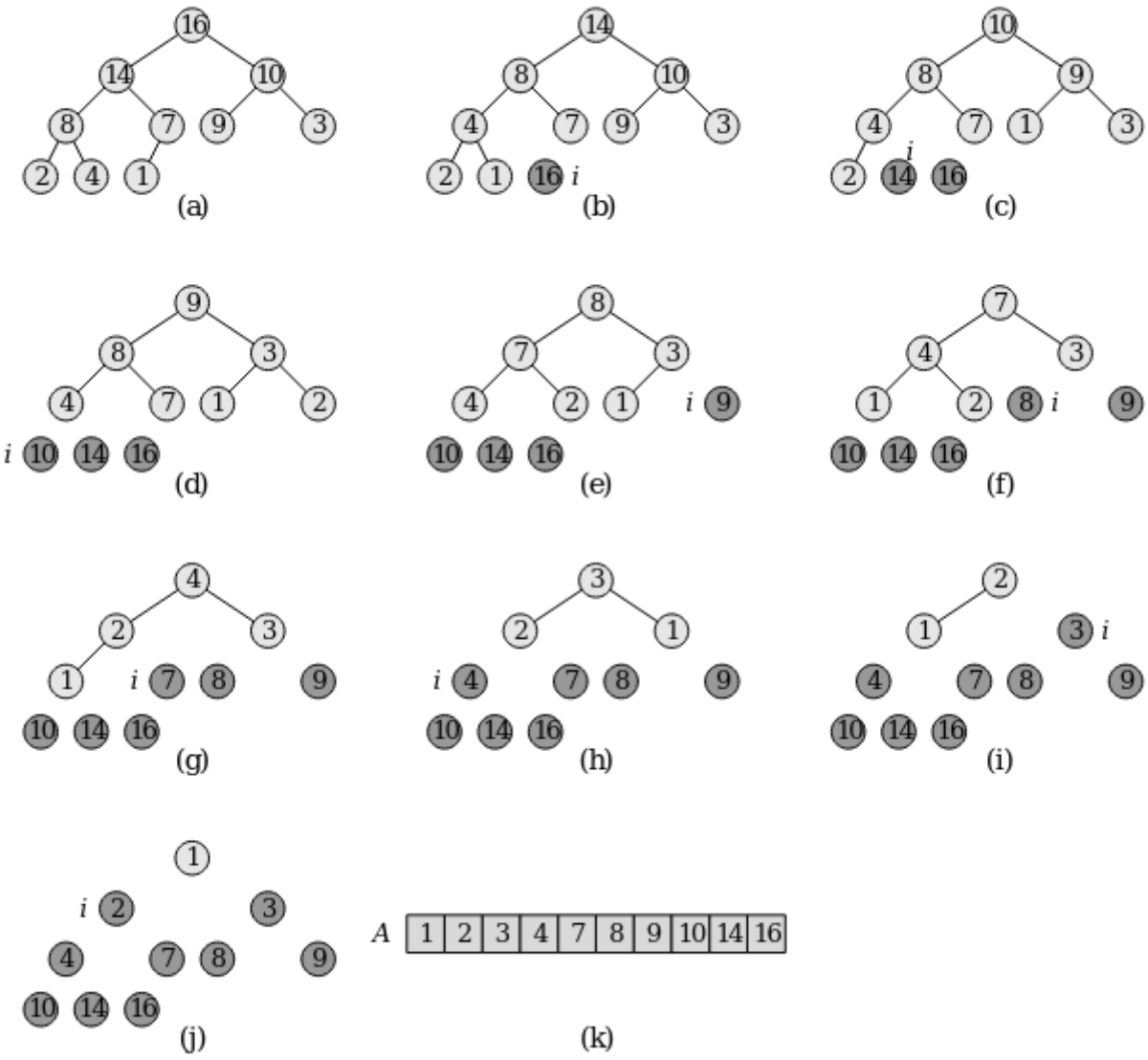
Legrosszabb eset: Teljesen fordítva rendezett tömb az input: `[5, 4, 3, 2, 1]`. Ekkor minden `beillesztendo` elemre vissza kell lépkedni a tömb elejéig.

Kupacrendezés

```
const kupacRendezes = (A: number[]) => {
    maximumKupacotEpit(A); // Helyben kupacosítja
    for (let i = A.length - 1, i >= 1; i--) {
        csere(A[1], A[i]);
        kupacMeret[A]--;
        maximumKupacol(A, 1);
    }

    return A;
}
```

Az input tömböt először **maximum-kupaccá** kell alakítani. Ekkor tudjuk, hogy a legnagyobb elem a gyökérben van, így ezt berakhatjuk az éppenvizsgált pozícióra (`csere(A[1], A[i])`). Ez után már csak csökkentenünk kell a kupac méterét, hiszen nem akarjuk mégegyszer a gyökérben az `A[i]`-t. Végezetül helyre kell állítanunk a kupac-tulajdonságot egy `maximumKupacol(A, 1)` hívással. (A 2. paraméter azt mondja meg, melyik csúcsbtól lefelé szeretnénk helyreállítani, jelen esetben az 1-es, hiszen pont azt a pozíciót rontottuk el, amikor cseréltünk. Tehát az egész kupacot helyreállítjuk.)



Futásidő	Tárigény (össz ~ inputon kívül)
$O(n * log(n))$	$O(n) \sim O(1)$

Gyorsrendezés

Összefésülő rendezéshez hasonlóan oszd-meg-és-uralkodj algoritmus

- **Felosztás**: Az `A[p..r]` tömböt, két (esetleg üres) `A[p..q-1]` és `A[q+1..r]` résztömbre osztjuk, hogy az `A[p..q-1]` minden eleme kisebb, vagy egyenlő `A[q]`-nál, és `A[q]` kisebb vagy egyelő `A[q+1..r]` minden eleménél. A `q` index kiszámítása része ennek a felosztó eljárásnak.
- **Uralkodás**: Az `A[p..q−1]` és `A[q+1..r]` résztömböket a gyorsrendezés rekurzív hívásával rendezzük.
- **Összevonás**: Mivel a két résztömböt helyben rendeztük, nincs szükség egyesítésre: az egész `A[p..r]` tömb rendezett.

```
const feloszt = (A: number[], p: number, r: number) => {
  const x = A[r];
  let i = p - 1;
  for (let j = p; j <= r - 1; j++) {
    if (A[j] <= x) {
      i++;
      [A[i], A[j]] = [A[j], A[i]];
    }
  }
  [A[r], A[i + 1]] = [A[i + 1], A[r]];
  return i + 1;
};
```

```
const _gyorsRendezes = (A: number[], p: number, r: number) => {
  if (p < r) {
    const q = feloszt(A, p, r);
    _gyorsRendezes(A, p, q - 1);
    _gyorsRendezes(A, q + 1, r);
  }
  return A;
};

const gyorsRendezes = (A: number[]) => _gyorsRendezes(A, 0, A.length - 1);
```

Futásidő	Tárigény
$O(n^2)$	$O(n)$

> Fontos, hogy az eljárás teljesítménye függ attól, hogy a felosztások mennyire ideálisak. Valószívűségi alapon a vátható rekurziós mélység $O(logn)$, ami mivel egy hívás futásideje $O(n)$, így az átlagos futásidő $O(n * logn)$. A gyakorlat azt mutatja, hogy ez az algoritmus jól teljesít.

> Lehet úgy implementálni, hogy $O(logn)$ tárigénye legyen, ez egy helyben rendező, farok-rekurzív ejlárás.

Összehasonlító rendezések teljesítményének alsó korlátja

Minden összehasonlító rendező algoritmus legrosszabb esetben $\Omega(n * logn)$ összehasonlítást végez.

> Ez alapján pl. az összefésülő, vagy a kupac rendezés **aszimptotikusan optimális**.

> Eddigi algoritmusok mind összahasonlító rendezések voltak, a kövezkező már nem az.

Ezt döntési fával lehet bebizonyítani, aminek belső csúcsai meghatároznak két tömbelemet, amiket épp összehasonlítunk, a levelek pedig hogy az oda vezető összehasonlítások milyen sorrendhez vezettek. Nem konkrét inputra írható fel döntési fa, hanem az algoritmushoz. Így ennek a fának a legrosszabb esetben vett magassága lesz az algoritmus futásidejének felső korlátja.

Leszámoló rendezés

Feltételezzük, hogy az összes bemeneti elem 0 és k közé esik.

Minden lehetséges bemeneti elemhez megszámoljuk, hányszor fordul elő az inputban.

Majd ez alapján azt, hogy hány nála kisebb van.

Ez alapján már tudjuk, hogy az egyes elemeknek hova kell kerülni. Mert ha pl 5 elem van, ami kisebb, vagy egyenlő, mint 2, akkor tudjuk, hogy az 5. pozición 2-es kell, hogy legyen.

```
const leszamoloRendezes = (A: number[], k: number) => {
  const C = [...new Array(k + 1)].map(() => 0)
  A.forEach(szam => {
    C[szam]++
  })
  // Itt a C-ben azon elemek száma van, aminek értéke i

  for (let i = 1; i < C.length; i++) {
    C[i] += C[i - 1]
  }
  // Itt C-ben i indexen azon elemek száma van, amik értéke kisebb, vagy egyenlő, r

  const B = [...new Array(A.length)] // B egy A-val egyező hosszú tömb

  for (let i = A.length - 1; i >= 0; i--) {
    B[C[A[i]] - 1] = A[i]
    C[A[i]]--
  }

  return B
}
```

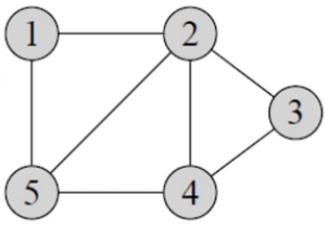
Futásidő	Tárigény

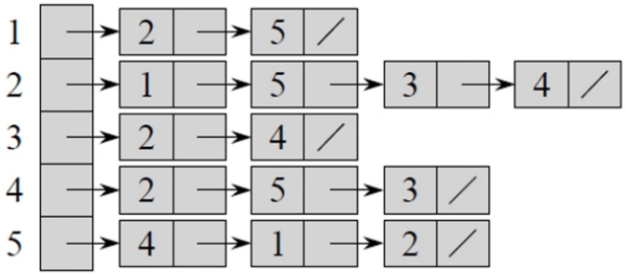
$\Theta(k+n)$	$\Theta(2n)$

> A gyakorlatban akkor használjuk, ha $k = O(n)$, mert ekkor a futásidő $\Theta(n)$

Gráfalgoritmusok

Gráfok ábrázolása: **éllista** vagy **szomszédsági mátrix**





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

Szélességi keresés

Gráf bejárására szolgál.

A bejárás során kijelöl egy "szélességi fát", ami egy kiindulási csúcsból indulva mindig az adott csúcsból elérhető csúcsokat reprezentálja.

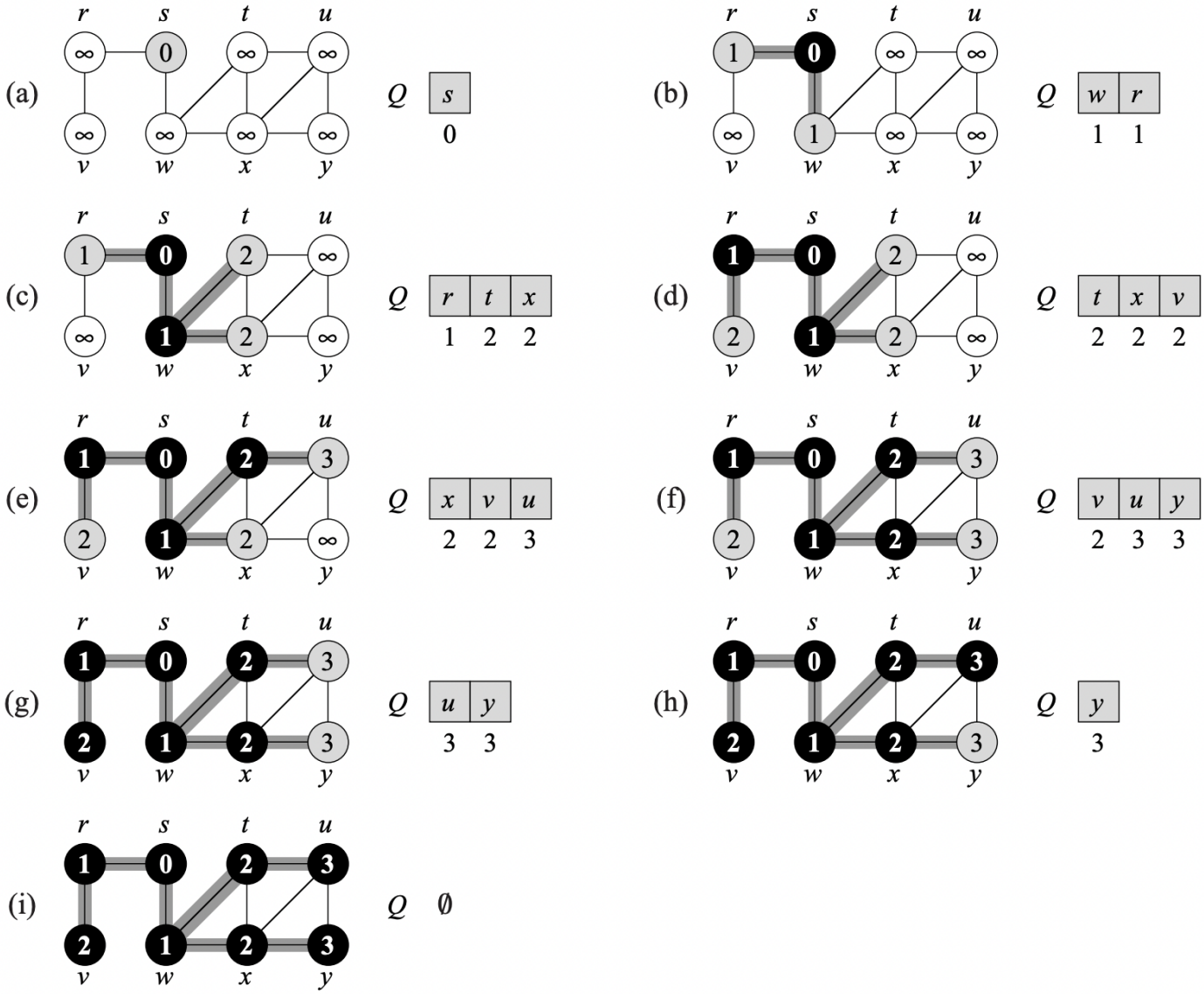
Amilyen távol van a kiindulási csúcstól egy csúcs, az olyan mélységen helyezkedik el ebben a fában.

Irányított, irányítatlan gráfog esetén is alkalmazható.

A csúcsok távolsága alapján kalad a bejárás (a kijelölt kezdeti csúcstól), minden k távolságra levő csúcsot elérünk az előtt, hogy egy $k+1$ távolságra levőt elérnénk.

Az algoritmus színezi a csúcsokat, ezek a színek a következőket jelentik:

- **fehér**: Kiindulási szín, egy ilyen színű csúcsot még nem értünk el.
- **szürke**: Elért csúcs, de még van fehér szomszédja.
- **fekete**: Elért csúcs, és már minden szomszédja is elért (vagy szürke vagy fekete).




```
// A G a gráf, s a kiindulási csúcs
szelessegiKereses(G, s) {
    for G grás minden nem s csúcsára {
        szín[csucs] = "fehér"
    }
    szín[s] = "szürke"
    d[s] = 0 // Távolság s-től
    szülő[s] = null
    Q = [] // Üres SOR
    sorba(Q, s)
    while Q nem üres {
        u = sorból(Q)
        for u minden v szomszédjára {
            if (szín[v] === "fehér") {
                szín[v] = "szürke"
                d[v] = d[u] + 1
                szülő[v] = u
                sorba(Q, v) // Tovább feldolgozzuk majd neki a szomszédjait
            }
        }
        szín[u] = "fekete" // Itt már végigmentünk minden szomszédján
    }
}
```

Futásidő

- Minden csúcsot egyszer érintünk csak, ez V db csúcs.
- Sorba, és sorból $O(1)$, így a sorműveletek összesen $O(V)$.
- Szomszédsági listákat legfeljebb egyszer vizsgáljuk meg, ezek össz hossza $\theta(E)$, így összesen $O(E)$ időt fordítunk a szomszédsági listák vizsgálására.
- Az algoritmus elején a kezdeti értékadások ideje $O(V)$.
- Összesített futásidő: $O(E+V)$

Mélységi keresés

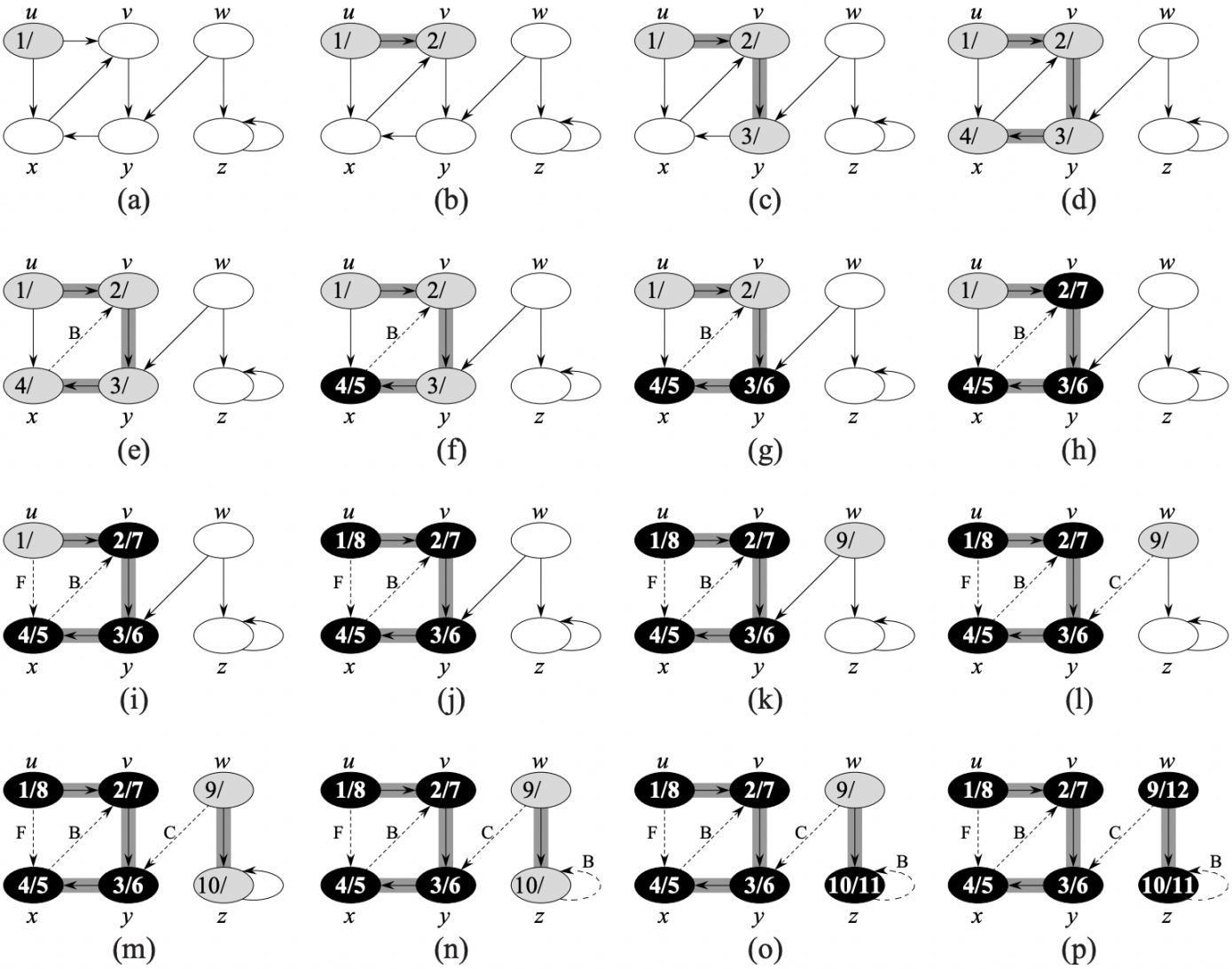
Addig megy a kivezető élek mentén, ameddig tud, majd visszafele indulva minden érintett csúcs kivezető élein addig megy mélyre, amíg lehet.

Ugyan azokat a színekez használja a csúcsok színezésére, mint a szélességi keresés.

Minden csúcshoz feljegyzi, hogy mikor (hány lépés után) érte el, és hagyta el azt.

```
melysegiKereses(G) {
    for G minden u csúcsára {
        szín[u] = "fehér"
        szülő[u] = null
    }
    idő = 0
    for G minden u csúcsára {
        if (szín[u] === "fehér") {
            melysegiBejaras(u)
        }
    }
}

melysegiBejaras(u) {
    szín[u] = "szürke"
    idő++
    d[u] = idő // Ekkor értük el
    for u minden v szomszédjára {
        if (szín[v] === "fehér") {
            szülő[v] = u
            melysegiBejaras(v) // Azonnal már indulunk is el a talált csúcsból
        }
    }
    szín[u] = "fehete"
    ido++
    f[u] = ido // Ekkor hafytuk el
}
```



Futásidő

A melysegiKereses() futásideje a melysegiBejaras() hívástól eltekintve $\Theta(V)$. A melysegiBejaras() hívások össz futásideje $\Theta(E)$, mert ennyi a szomszédsági listák összesített hossza. Így a futásidő $O(E+V)$

> A futásidő azért lesz additív mingkét esetben, mert a szomszédsági listák össz hosszára tudjuk mondani, hogy $\Theta(E)$. Lehet, hogy ezt egyszerre nézzük végig, lehet, hogy eloszlatva, de **összessen** ennyi szomszédot vizsgál meg például a mélységiBejárás().

Minimális feszítőfák

Cél: megtalálni éleknek azon **körmentes** részhalmazát, amely élek mentén **minden csúcs összeköthető,** és az élek **összesített súlya** legyen a **lehető legkisebb**.

Az így kiválasztott élek egy fát alkotnak, ez a **feszítőfa**.

Két **mohó** algoritmus: **Prim**, **Kruskal**

Kruskal

A gráf csúcsait diszjunkt halmazokba sorolja. Kezdetben minden csúcs 1-1 egy elemú csúcs.

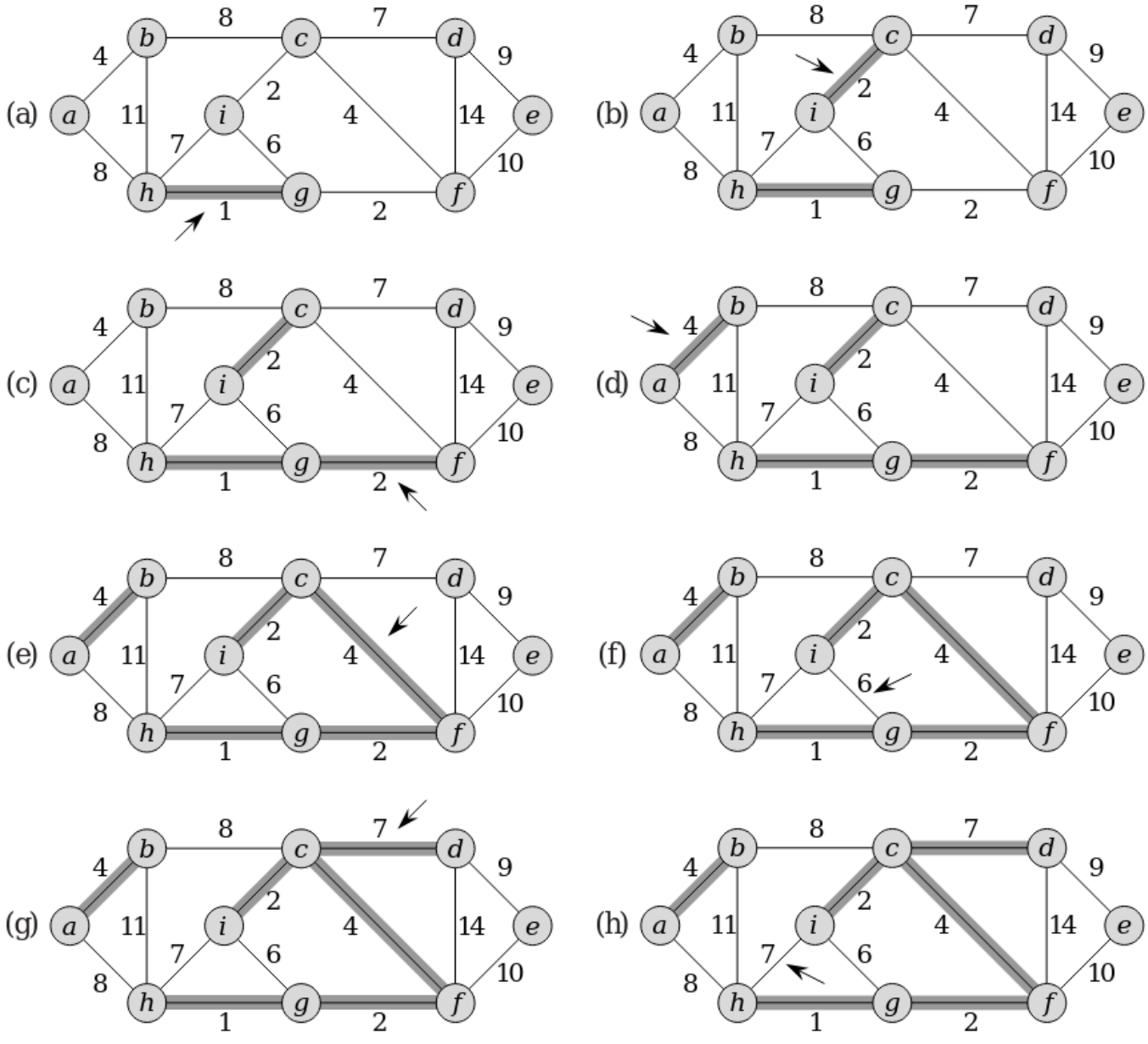
> Erre van speciális diszjunkt-halmaz adatszerkezet

Minden iterációban beveszi a legkisebb súlyú élet, aminek végpontjai különböző halmazokban vannak.

Ez által egy erdőt kezel, mit a végére egy fává alakít. Ez lesz a feszítőfa.

```
kruskal(G, w) { // Az élsúlyokat megadó függvény
    A = 0
    for minden v csúcsra {
        halmaztKeszit(v)
    }
    for minden (u, v) élre, az élsúlyok szerin növekvő sorrendben {
        if halmaztKeres(u) != halmaztKeres(v) {
            A = A unió { (u, v) }
            egyesít(u, v)
        }
    }
}
```

`halmaztKeszit`, `halmaztKeres` és `egyesít` a diszjunkt halmazokat kezelő függvények.



######## Futásidő

Az élek rendezése $O(E * logE)$.

A halmaz műveletek a kezdeti értékadásokkal együtt $O((V + E) * \alpha * (V)$. Ahol az α egy nagyon lassan növekvő függvény, a diszjunkt-halmaz adatszerkezet jasátossága. Mivek összefüggő gráf esetén $O(|E| \geq |V| + 1)$, így a diszjunkt-halmaz műveletek $O((E) * \alpha * (V))$ idejűek. $\alpha(|V|) = O(logE)$ miatt $O(E * logE)$.

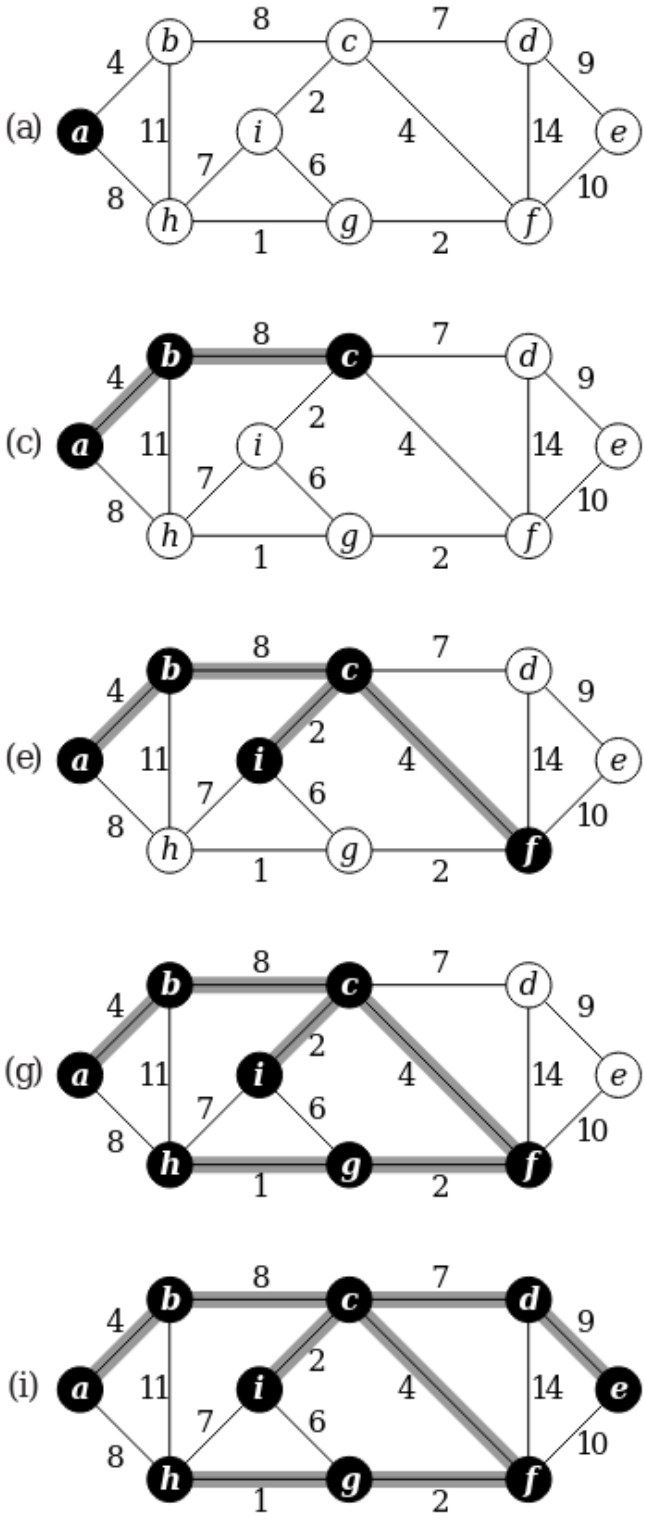
Így a teljes futásidő $O(E * logE)$.

Prim algoritmus

A Kruskallal ellentétben folyamatosan egy darab fát kezel, ezt növeli az iterációkban.

Egy megadott kiindulási csúcsból indulva minden iterációban hozzávesszük azt a csúcsot, amit a legkisebb súlyú él köt a meglévő fához.

```
prim(G, w, r) { // Az élsúlyokat megadó függvény
    for minden v csúcsra {
        kulcs[v] = Végtelen
        szülő[v] = null
    }
    kulcs[r] = 0
    Q = G csúcsai // Prioritási sor kulcs[] szerint minimális
    while Q nem üres {
        u = kiveszMin(Q)
        for u minden v szomszédjára {
            if v eleme Q, és w(u, v) < kulcs[v] {
                szülő[v] = u
                kulcs[v] = w(u, v)
            }
        }
    }
}
```



######## Futásidő

Bináris minimum kupac megvalósítással:

Kezdeti értékadások: $O(V)$

Egy db kiveszMin művelet: $O(logV)$. Összesen: $O(V * logV)$, mivel V-szer fut le a ciklus.

Belső for ciklus $O(E)$-szer fut, mivel szomszédsági listák hosszainak összege: $O(2|E|)$. (Ez megintcsak additív, nem kell a külső ciklussal felszorozni, mert a szomszédsági listák alapján tudjuk, hogy ennyiszer fog maximum összesen lefutni.) Ezen a cikulson belül a Q-hoz tartozás vizsgálata konstans idejű, ha erre fenntartunk egy jelölő bitet. A kulcs-nak való értékadás valójában egy kulcsotCsökkent művelet, ami $O(logV)$ idejű.

Agy tehát az összesített futásidő: $O(VlogV + ElogV) = O(ElogV)$.

> Fibonacchi-kupaccal gyorsítható az algoritmus, ekkor a kiveszMin $O(logV)$-s, kulcsotCsökkent $O(1)$-es, teljes futásidő: $O(E + V * logV)$

Legrövidebb utak

Lehetséges problémák:

- **Adott csúcsból induló legrövidebb utak problémája**: Egy adott kezdőcsúcsból meg szeretnénk találni minden másik csúcshoz vezető legrövidebb utat.
- **Adott csúcsba érkező legrövidebb utak problémája**: Minden csúcsból egy adott csúcsba. Ugyan az, mint az előbbi, ha az élek irányát megfordítjuk.
- **Adott csúcspár közti legrövidebb út problémája:** Ha az elsőt megoldjuk, ezt is megoldottuk. Nem ismert olyan algoritmus, ami aszimptotikusan gyorsabban megoldaná ezt a feladatot, de az elsőt nem.
- **Összes csúcspár közti legrövidebb utak problémája**: Ez persze megoldható lenne az elsővel, ha minden csúcsból elindítjuk, de ennél léteznek gyorsabb megoldások.

Optimális részstruktúra: Azt jelenti jelen esetben, hogy két csúcs közti legrövidebb út magában foglalja sokszor másik két csúcs közti legrövidebb utat. Az algoritmusok ezt használják ki.

Negatív súlyú élek: Lehetnek, de a gráf nem tartalmazhat **negatív összsúlyú kört**. Ugyanis ekkor nem definiált a legrövidebb út, hiszen a körön mégegyszer végig haladva mindig kisebb súlyú utat kapunk.

Kör a legrövidebb útban: Negatív összsúlyú tehát nem lehet, mert ekkor maga a feladat nem definiált. **Pozitív összsúlyú sem lehet**, hiszen ekkor jobban járnánk, ha nem járnánk be a kört. **Nulla összsúlyúnak pedig nincsen értelme**, hogy szerepeljen legrövidebb útban, hiszen ekkor ugyan annyi az összsúly a kör megtétele nélkül is. Tehát általánosságban feltételezhetjük, hogy a **legrövoidebb út nem tartalmaz kört**.

Két függvény, amit használni fognak az algoritmusok:

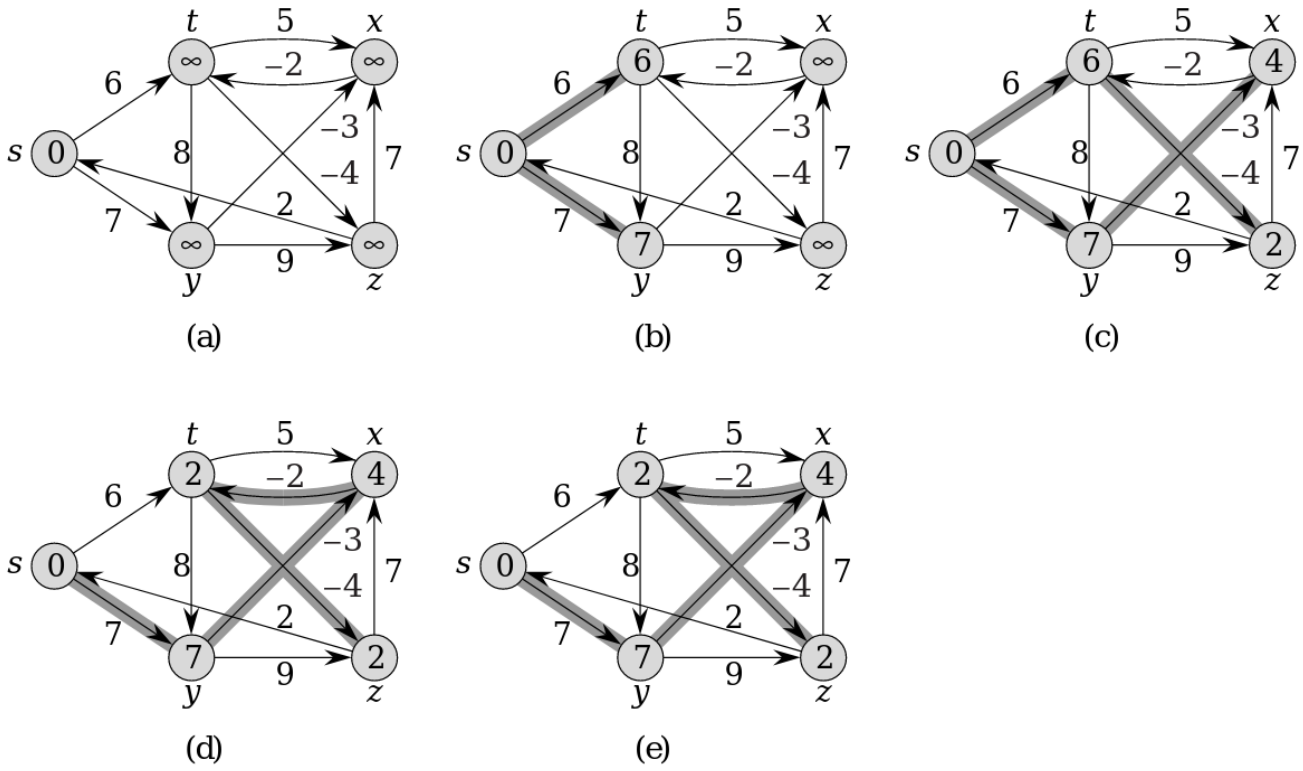
```
egyForrasKezdoertek(G, s) { // Kezdőértékek beállítása, ha egy csúcsból indul
    for minden v csúcsra {
        f[v] = Végtelen
        szülő[v] = null
    }
    d[s] = 0
}


közelít(u, v, w) { // (u, v) él alapján v távolságának frissítése (ha u-ból jöve k
    if d[v] > d[u] + w(u, v) {
        d[v] = d[u] + w(u, v) // A d[v] becslést csökkenti
        szülő[v] = u
    }
}
```

Bellman-Ford algoritmus

Lehetnek negatív élek, ha van negatív összsúlyú él, azt felismeri az algoritmus, jelzi azzal, hogy hamissal tér vissza.

```
bellmanFord(G, w, s) {
    egyForrasKezdoertek(G, s)
    for i = 1 to |V[G]| - 1 {
        for minden (u, v) élre {
            közelít(u, v, w)
        }
    }
    for minden (u, v) élre { // Itt ellenőrzi, hogy volt-e negatív kör
        if d[v] > d[u] + w(u, v) {
            return false
        }
    }
    return true
}
```



######## Futásidő

$O(V * E)$ hiszen a kezdőértékek beállítása $\Theta(V)$, az egymásba ágyazott for ciklus $O(V * E)$, a második ciklus pedig $O(E)$.

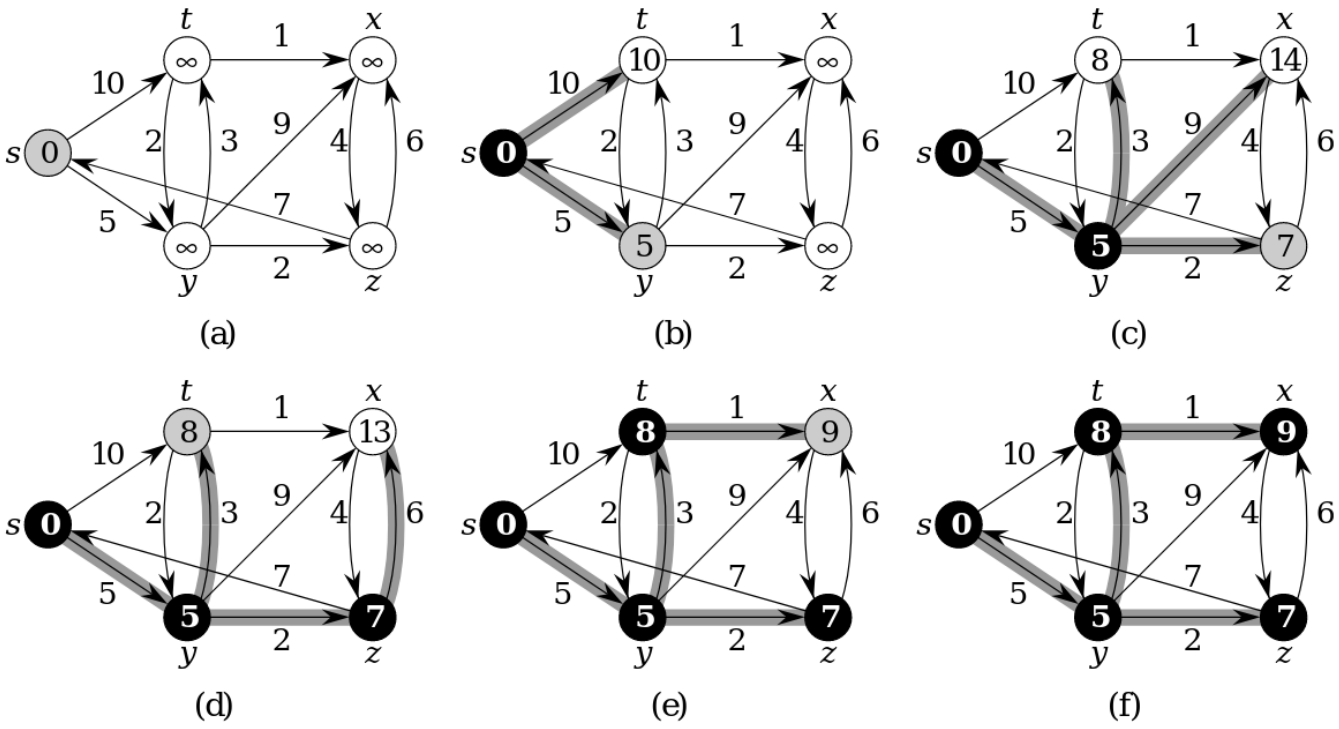
Dijkstra algoritmusa

Nemnegatív élsúlyok esetén működik.

S halmaz: Azon csúcsok kerülnek bele, amikhez már meghatározta a legrövidebb utat a kezdőcsúcsból.

```
dijkstra(G, s) {
    egyForrasKezdoertek(G, s)
    S = üresHalmaz
    Q = V[G] // Q minimum prioritási sor
    while Q nem üres {
        u = kiveszMin(Q)
        S = S unió { u }
        for u minden v szomszádjára {
            közelít(u, v, w)
        }
    }
}
```

A Q sorban azok a csúcsok vannak, amik nincsenek S-ben, tehát még nem tudjuk a hozzájuk vezető legrövidebb utat. A sort a d érték szerint azaz az ismert legrövidebb út szerint indexeljük.



######## Futásidő

Minden csúcs pontosan egyszer kerül át az S halmazba, emiatt amikor szomszédokat vizsgálunk, azt minden csúcsra egyszer tesszük meg, ezen szomszédok vizsgálata összesen $O(E)$-szer fut le, mert ennyi a szomszédsági listák össz hossza. Így a közelít, és ez által a `kulcsotCsökkent` művelet legfejlebb $O(E)$-szer hívódik meg.

Az összesített futásidő nagyban függ a **prioritási sor implementációtól**, a legegyszerűbb eset, ha egy **tömbbel implementáljuk**. Ekkor a `beszúr` és `kulcsotCsokkent` műveletek $O(1)$-esek, a `kiveszMin` pedig $O(V)$, mivel az egész tömbön végig kell menni. Így a teljes futásidő $O(V^2 + E)$. **Ritkább gráfok esetén gyorsítható** az algoritmus **bináris kupac** implementációval, és látalánossagban gyorsítható fibonacchi kupaccal.

Floyd-Warshall algoritmus

Dinamikus programozási algoritmus legrövidebb utak **minden csúcspárra** problémára.

Lehetnek negatív élsúlyok, de negatív összsúlyú körök nem.

Az algoritmus lényege, hogy dinamikus programozással haladunk, egyre több csúcsot használunk fel, és azt figyeljük, hogy a két csúcs között vezető úton jobb eredményt érnénk-e el, ha az adott iteráció csúcsán keresztül mennénk.

Ez a következő rekurziós képlettel írható fel:

$$d_{ij}^{(k)} = \begin{cases} w_{ij}, & \text{ha } k = 0, \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}), & \text{ha } k \geq 1. \end{cases}$$

```
floydWarshall(W) { // W szomszédsági mártix
    n = sorokSzama(W)
    D(0) = W
    for k = 1-től n-ig { // Ezt vizsgáljuk mindig majd, mint köztes csúcs
        for i = 1-től n-ig {
            for j = 1-től n-ig {
                d(k)[i, j] = min(
                    d(k - 1)[i, j],
                    d(k - 1)[i, k] + d(k - 1)[k, j]
                )
            }
        }
    }
}
```

A belső értékadás magyarázata: A k. iterációban a legrövidebb út, ami i-ből j-be vezet, az vagy a már megtalált k - 1-edik iterációbeli eredmény, vagy a az előzőz iterációbeli út i-ből k-ba, plusz k-ból j-be, azaz **felhasználjuk-e a k-t, mint egy köztesen érintett csúcsot**.

######## Futásidő

A három for ciklus határozza meg, mert annak a magja $O(1)$-es, így a futásidő $\Theta(n^3)$, ahol n a sorok száma.

2. Elemi adatszerkezetek, bináris keresőfák, hasító táblázatok, gráfok és fák számítógépes reprezentációja

Az **adatszerkezet** adatok tárolására, és szervezésére szolgáló módszer, amely lehetővé teszi a hatékony hozzáférést és módosítést.

Algoritmushoz válasszuk ki az adatszerkezetet. Előfordulhat, hogy az algoritmus a megfelelő adatszerkezeten alapul.

Absztrakt adatszerkezet: műveletek által definiált adaszerkezet, nem konkrét implementáció.

Adatszerkezetek: Absztrakt adatszerkezetek konkrét megvalósításai. Általában egyes implementációk egyes műveleteket gyorsabban, míg másokat lassaban tudnak végrehajtani. Ez alapján kell az algoritmushoz kiválasztani a megfelelőt.

> Absztrakt adatszerkezetek olyanok, mint **interfészek**, az adatszerkezetek pedig azt implementáló **osztályok**.

Listák

Absztrakt adatszerkezet.

Benne az adatok lineárisan követik egymást, egy kulcs többször is előfordulhat benne.

Művelet	Magyarázat
`ÉRTÉK(H, i)`	`i`. pozíción (index-en) a kulcs értékének visszaadása
`ÉRTÉKAD(H, i, k)`	`i`. pozíción levő értéknek a `k` érték értéküladása
`KERES(H, k)`	A `k` kulcs (érték) megkerekéke a listában, indexének visszaadása
`BESZÚR(H, k, i)`	Az `i`-edik pozíctó után a `k` beszúrása
`TÖRÖL(H, k)`	Első `k` értékű elem törlése

Közvetlen elérésű lista

Összefüggő memóriaterületet foglalunk le, így minden index közvetlen elérésű.

Művelet	Futásidő
`ÉRTÉK(H, i)`	$O(1)$
`ÉRTÉKAD(H, i, k)`	$O(1)$
`KERES(H, k)`	$O(n)$
`BESZÚR(H, k, i)`	$O(n)$
`TÖRÖL(H, k)`	$O(n)$

> Beszúrásnál újra kellhet allokálni egyel nagyobb emmóriaterületet.

> Jellemzően úgy implementáljuk, hogy definiálunk egy **kapacitást**, és amikor kell, akkor eggyivel allokálunk többet az új memóriaterületen. Illetve jellemzően azt is definiáljuk, hogy mikor kell zsugorítani a területet, azaz hány üresen maradó cella esetén (nem lyukak! az nem lehet, csak a terület végén levő üres cellák) allokáljunk kevesebb területet.

Előnye: O(1)-es indexelés.

Hártánya: Módosító műveletek lassúal, egy nagy memóriablokk kell.

Láncolt lista

Minden kulcs mellett tárolunk egy mutatót a következő, és egy mutatót a megelőző elemre.

Egyszeresen láncolt lista: csak a következőre tárolunk mutatót.

Kétszeresen láncolt lista: Következőre, előzőre is tárolunk mutatót.

Ciklikus lista: Utolsó elem rákövetkezője az első elem, első megelőzője az uolsó elem.

Őrszem / fej: Egy NULL elem, ami mindig a lista eleje.

Művelet	Futásidő
`ÉRTÉK(H, i)`	$O(n)$
`ÉRTÉKAD(H, i, k)`	$O(n)$
`KERES(H, k)`	$O(n)$
`BESZÚR(H, k, i)`	$O(1)$
`TÖRÖL(H, k)`	$O(1)$

> Beszúrás, és törlés valójában $O(n)$. Csak akkor $O(1)$, ha már a megfelelő pozíción vagyunk, azaz már tudjuk, melyik mutatókat kell átírni.

Előnye: Nem egy nagy összefüggő memória blokk kell.

Hártánya: Nem lehet gyorsan indexelni. Tárigény szempontjából rosszabb, minden kulcs mellett tárolunk legalább egy mutatót.

Verem

Lista, amiben csak a legutoljára beszúrt elemet lehet kivenni. (**LIFO**)

Emiatt a speciális művelet végzés miatt gyorsabb, mint a sima lista.

Alkalmazásokra pl.: Függvényhívások veremben, undo-redo, böngésző előzmények.

Verem megvalósítás fix méretű tömbbel

Fenntartunk egy mutatót a verem tetejére, eddig van feltöltve a lefoglalt memóriaterület. (A verem alja a 0. index.)

```
üresVerem(V) {
    return tető[V] == 0
}
```

```
verembe(V, x) {
    tető[V]++ // Tető mutató frissítése, hiszen egyel több elem lesz
```

```
    V[tető[V]] = x
}
```

```
veremből(V) {
    if üresVerem(V) {
        throw Error("alulcsordulás")
    } else {
        tető[V]--
        return V[tető[V] + 1] // Ez az index nincs felszabadítva, vagy átírva, egy
    }
}
```

Mind a 3 művelet $O(1)$-es, hiszen csak indexeléseket, értékadásokat tartalmaznak.

> Hasonlóan a tömbbel megvalósított listához, itt is érdemes lehet kapacitást meghatározni.

Sor

Mindig a legelőször beszúrt elemet lehet kivenni. (**FIFO**)

Lefoglalunk egy valamekkora egybefüggő memória szegmenst, de nem mindig használjuk az egészet. Két mutatót tartunk fent, a `fej` és a `vége` mutatókat, ezek jelölik, hogy éppen mekkora részét használjuk a lefoglalt területnek sorként.

```
sorba(S, x) {
    S[vége[S]] = x // A vége egy üres pozícióra mutat alapból, ezért növeljük utól
    if vége[S] = hossz[S] {
        vége[S] = 1 // Ekkor "körvefordult" a sor a lefoglalt memóriaterületen.
    } else {
        vége[S]++
    }
}
```

```
sorból(S) {
    x = S[fej[S]] // A fej mutat a sor "elejére", azaz a legrégebben betett elemre
    if fej[S] == hossz[S] {
        fej[S] = 1 // Ekkor "körvefordult" a sor a lefoglalt memóriaterületen.
    } else {
        fej[S]++
    }
}
```

Mind a két művelet $O(1)$-es, hiszen csak indexeléseket, értékadásokat tartalmaznak.

Prioritási sor

Absztrakt adatszerketet.

Nem a kulcsok beszúrásának sorrendje határozza meg, mit lehet kivenni, hanem mindig a maximális (vagy minimális) kulcsú elemet tudjuk kivenni.

Művelet	Magyarázat
`BESZÚR(H, k)`	Új elem beszúrása a prioritási sorba
`MAX(H)`	Maximális kulcs értékének visszaadása
`KIVESZ-MAX(H)`	Maximális kulcsú elem kivétele (vagy minimális)

Kupac

Hatékony **prioritási sor megvalósítás**.

A kupac egy **majdnem teljes bináris fa**, amiben minden csúcs értéke legalább akkora, mint a gyerekeié, ezáltal a maximális (minimális) kulcsú elem a gyökérben van.

Majdnem teljes bináris fa alatt azt értjük, hogy a fa legmélyebb szintjén megengedett, hogy balról jobbra haladva egyszer csak már ne álljon fenn a bináris fa tulajdonság.

Tömbös megvalósítás

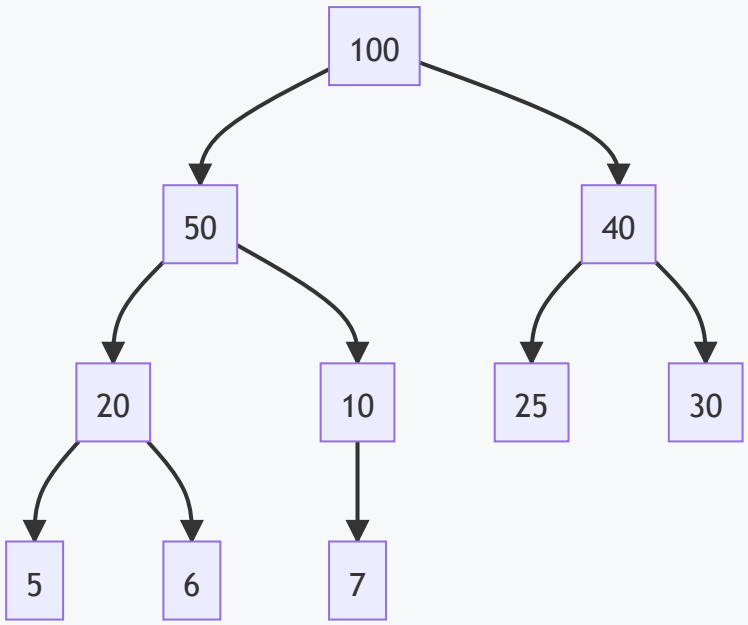
Egybefüggő memóriaterületen van a teljes kupac.

A **szülő**, a **bal gyerek**, és a **jobb gyerek** gyorsan számolható a tömb indexelésével.

```
szülő(i) { // i indexő elem szülője
    return alsoEgeszResz(i / 2)
}
```

```
balGyerek(i) { // i indexű elem bal gyereke
    return 2i
}
```

```
jobbGyerek(i) { // i indexű elem jobb gyereke
    return 2i + 1
}
```



Ennek a kupacnak a tömbös reprezentációja:

```
[100, 50, 40, 20, 10, 25, 30, 5, 6, 7]
```

######## Kupactulajdonság fenntartása

Garanálnunk kell, hogy az egyes beszúrások, kivételek után a kupacra jellemző tulajdonságok fennmaradnak.

A tulajdonság fenntartására ez a függvény fog felelni:

```
maximumKupacol(A, i) {
    l = balGyerek(i)
    r = jobbGyerek(i)
    if l <= kupacMéret[A] és A[l] > A[i] { // l <= kupacMéret[A] ellenőrzés csak a
        legnagyobb = l
    } else {
        legnagyobb = i
    }
    if r <= kupacMéret[A] és A[r] > A[i] { // r <= kupacMéret[A] ellenőrzés csak a
        legnagyobb = r
    }
    if legnagyobb != i {
        csere(A[i], A[legnagyobb])
        maximumKupacol(A, legnagyobb)
    }
}
```

Tehát a vizsgált indexű elem et összehasonlítjuk a gyerekeivel, és ha valamelyik nagyobb, akkor azzal kicseréljük, és rekurzívan meghívjuk rá a `maximumKupacol()`-t, mert lehet, az új

szülőjénél/gyerekénél is nagyobb.

`maximumKupacol()` futásideje $O(logn)$, mert ennyi a majdnem teljes bináris fa mélysége, és legrosszabb esetben az egészen végig kell lépkedni.

####### Maximum lekérése

A prioritási sor `MAX(H)` függvényének megvalósítása egyszerű, csak vissza kell adnunk a tömb első elemét, ami a kupac gyökere.

```
kupacMaximuma(A) {
    return A[1]
}
```

####### Maximum kivétele

Ilyenkor az történik, hogy a kupac utolsó elemét áthelyezzük a gyökérbe, és a gyökérből indulva helyreállítjuk a kupac tulajdonságot, "lekupacoljuk" az elemet.

```
kupacbólKiveszMaximum(A) {
    if kupacMéret[A] < 1 {
        throw Error("kupacméter alulcsordulás")
    }
    max = A[1]
    A[1] = A[kupacMéret[A]]
    kupacMéret[A]-- // Méter csökkentése, az érték a memóriában marad, csak nem ér
    maxumimKupacol(A, 1) // Mivel beszúrtuk ide az utolsó elemet, helyre kell állí
    return max
}
```

####### Beszúrás

Új elem beszúrása egyszerű, csak szúrjuk be a kupac végére, és onnan kiindulva végezzünk egy helyreállítást, ezzel az új elemet a helyére "felkupacolva".

```
kupacbaBeszur(A, x) {
    kupacMéter[A]++
    A[kupacMéret[A]] = x
    maximumKupacol(A, kupacMéret[A])
}
```

####### Futásidők

Művelet	Futásidő
`BESZÚR(H, k)`	$O(logn)$

`MAX(H)`	$\Theta(1)$
`KIVESZ-MAX(H)`	$O(logn)$

Fák, és számítógépes reprezenzációjuk

Fa

- Összefüggő, körmentes gráf
- Bármely két csúcsát pontosan egy út köti össze
- Elsőfokú csúcsi: **levél**
- Nem levél csúcsai: **belső csúcs**

Bináris fa

- **Gyökeres fa**: Van egy kitűntetett gyökér csúcsa
- **Bináris fa**: Gyökeres fa, ahol minden csúcsnak legfeljebb két gyereke van.

Számítógépes reprezentáció

Csúcsokat, és éleket reprezentálunk.

Maga a fa objektumunk egy mutató a gyükérre.

Gyerek éllistás reprezentáció

```
class Node {
    Object key;
    Node parent;
    List<Node> children; // Gyerekek éllistája
}
```

Első fiú - apa - testvér reprezentáció

```
class Node {
    Object key;
    Node parent;
    Node firstChild;
    Node sibling;
}
```

Bináris fa reprezentációja

```
class Node {
    Object key;
    Node parent;
    Node left;
    Node right;
}
```

> Mindegyik esetben, ha nincs Node, akkor NULL-al jelezhetjük. Pl. a gyökér szülője esetében.

Bináris keresőfák

Absztrakt adatszerkezet a következő műveletekkel:

Művelet	Magyarázat
`KERES(T, x)`	Megkeresi a fában az `x` kulcsot, és visszaadja azt a csúcsot
`BESZÚR(T, x)`	Fába az `x` kulcs beszúrása
`TÖRÖL(T, x)`	Fából az `x` kulcsú csúcs törlése
`MIN(T)` / `MAX(T)`	A fa maximális, vagy minimális kulcsú csúcsának visszaadása
`KÖVETKEZŐ(T, x)` / `ELŐZŐ(T, x)`	A fában az `x` kulcsnál egyel nagyobb, vagy egyel kisebb értékű csúcs visszaadása

> A `T` a fa gyökerére mutató mutató.

> Cél: Minden művelet legalább $O(logn)$-es legyen

Bináris keresőfa tulajdonság

Egy x csúcs értéke annak a bal részfájában minden csúcsnál nagyobb vagy egyenlő, jobb részfájában minden csúcsnál kisebb vagy egyenlő.

Keresés

A bináris fa tulajdonságot kihasználva fa keresendő kulcsot hasonlítgatjuk a bal, jobb gyerekekhez, és ennek megfelelően lépünk jobbra / balra.

```
fábanKeres(x, k) {
    while x != NULL és k != kulcs[x] {
        if k < kulcs[x] {
            x = bal[x]
        } else {
            x = jobb[x]
        }
    }
    return x
}
```

Minimum / Maximum keresés

A minimum elem a "legbaloldali" elem

```
fábanMinimum(x) {
    while bal[x] != NULL {
        x = bal[x]
    }
    return x
}
```

A maximum elem a "legjobboldali" elem

```
fábanMaximum(x) {
    while jobb[x] != NULL {
        x = jobb[x]
    }
    return x
}
```

Következő / Megelőző

```
fábanKövetkező(x) {
    if jobb[x] == NULL {
        return fábanMinimum(jobb[x])
    }
    y = szülő[x]
    while y != NULL és x == jobb[y] {
        x = y
        y = szülő[y]
    }
    return y
}
```

Azaz, ha van jobb részfája a fának, amiben keresünk, akkor annak a mimimuma a rákövetkező, ha nincs, akkor pedig addig lépkedünk fel, amíg az aktuális csúcs a szülőjének bal gyereke nem lesz, ugyanis ekkor a szülő a rákövetkező.

TODO: Hasonlóan a megelőzőre.

Beszúr

```
fábaBeszúr(T, z) {
    y = null
```

```
    x = gyökér[T]
    while x != null {
        y = x
        if kulcs[z] < kulcs[x] {
            x = bal[x]
        } else {
            x = jobb[x]
        }
    }
    szülő[z] = y
    if y = null {
        gyökér[T] = z
    } else if kulcs[z] < kulcs[y] {
        bal[y] = z
    } else {
        jobb[y] = z
    }
}
```

Tehát megkeressük az új elem helyét, az által, hogy jobbra, balra lépkedünk, majd beszúrjuk a megfelelő csúcs alá jobbra, vagy balra.

Töröl

```
fábólTöröl(T, z) {
    if bal[z] == null vagy jobb[z] == null {
        y = z
    } else {
        y = fábanKövetkező(z)
    }

    if bal[y] != null {
        x = bal[y]
    } else {
        x = jobb[y]
    }

    if x != null { // x akkor null, ha y = fábanKövetkező(z)
        szülő[x] = szülő[y] // "átkötés"
    }

    if szülő[y] == null {
        gyökér[T] = x // Ha gyökérbe lett kötve az y, akkor ezt is frissítjük
    } else if y == bal[szülő[y]] {
        bal[szülő[y]] = x // "átkötés"
    } else {
        jobb[szülő[y]] = x // "átkötés"
    }

    if y != x {
        kulcs[z] = kulcs[y]
    }
```

```
        return y
    }
```

Levél törlése

Ha a kitörlendő csúcs egy levél, akkor egyszerűen kitöröljük azt, a szülőkénél a rá mutató mutatót `null`-ra állítjuk.

Egy gyerekes belső csúcs

Ebben az esetben a törlendő csúcs helyére bekötjük annak a részfáját ()amiből, mivel egy gyereke van, csak egy van).

Két gyerekes belső csúcs

Ebben az esetben a csúcs helyére kötjük annak a rákövetkezőjét. Mivel ebben az esetben van biztosan jobb gyereke, így a jobb gyerekének a minimumát fogjuk a helyére rakni (ami mivel egy levél, csak egyszerűen törölhetjük az eredeti helyéről).

Futásidők

Az összes művelet (`KERES` , `MAX / MIN` , `BESZÚR` , `TÖRÖL` , `KÖVETKEZŐ / ELŐZŐ`) $O(h)$-s, azaz a fa magasságával arányos. Ez alap esetben nem feltétlen olyan jó, de kiegyensúlyozott fák esetén jó, hiszen akkor $O(logn)$-es.

> Pl. AVL-fa, bináris kereső fa kiegyensúlyozott.

Halmaz

Absztrakt adatszerkezet.

Egy elem legfejlebb egyszer szerepelhet benne.

Művelet	Magyarázat
`TARTALMAZ(k)` (lényegében `KERES(k)`)	Benne van-e egy e `k` a halmazban?
`BESZÚR(K)`	Elem behelyezése a halmazba.
`TÖRÖL(K)`	Elem törlése a halmzból.

> Egyéb extra műveletek definiálhatóak, pl.: `METSZET` , `UNIÓ`

Közvetlen címzésű táblázat

Egy akkora tömb lefoglalása, mint amekkora a teljes érték univerzum mérete, és ha egy szám eleme a halmaznak, egyszerűen beírjuk ezt a megfelelő indexre.

Jó, mert nagyon gyors megoldás.

Viszont nagy probléma, hogy a tárigény az univerzum méretével arányos, nem pedig a ténylegesen felhasznált elemekkel.

Kis méretű univerzum esetén ajánlatos csak.

Szótár

Absztrakt adatszerkezet.

Egy halmaz elemeihez (kulcsok) egy-egy érték tartozik. Kulcs egyedi, érték ismétlődhet.

> dict, asszociatív tömb, map

Hasító tábla

Szótár, és halmaz hatékony megvalósítása.

Cél.: `TARTALMAZ`, `BESZÚR`, `TÖRÖL` műveletek legyenek gyorsak.

Hasító függvény

Kulcsok U univerzumának elemeit (lehetséges kulcsokat) képezi le a hasító táblázat **rés**eire.

Pl.: $h(k) = k \, mod \, m$

k a hasító táblázat mérete, azaz a **rések száma**.

Mivel az unicerzum, a lehetséges kulcsok száma nagyobb, mint réseké (különben csinálhatnánk tömbös megvalósítást), így elkerülhetetélen, hogy ürközések legyenek, azaz hogy a hasító függvény két kulcsot ugyan arra a résre képezzen le.

Ezeket az **ütközéseket fel kell oldani**.

Ütközésfeloldás láncolással

A résekben láncolt listák vannak.

Ha olyan helyre akarunk beszúrni, ahol már van elem, akkor a lista elejére szúrjuk be az újat (ez konstans idejű).

Keresés, törlés valamivel romlik, hiszen egy lsitán is végig kelhet menni.

Kitöltési tényező: $\alpha = \frac{n}{m}$ (**láncok átlagos hossza**)

m: rések száma

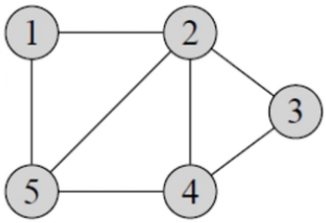
n: elemek a táblában

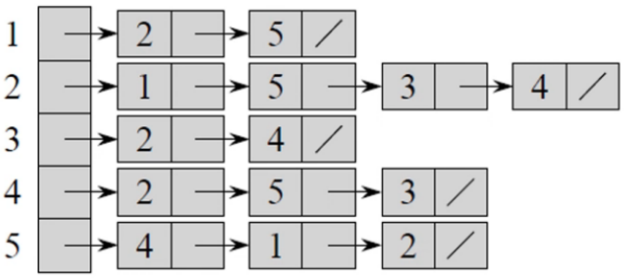
Egyszerű egyenletes hasítási feltétel: Minden elem egyforma valószínűséggel képződik le bármelyik résre.

Ha egy hasító függvény ezt biztosítja, akkor a keresések (mind sikeres, mind sikertelen) átlagos ideje (nem legrosszabb!) $\Theta(1 + \alpha)$

Ha tudjuk, mennyi elem lesz a táblában, akkor meg tudjuk választani a rések számát úgy, hogy az α egy konstans legyen, ekkor `KERES`, `TÖRÖL`, `BESZÚR` mind $O(1)$.

Gráfok számítógépes reprezentációja





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

- Csúcsok + élek halmaza
- Szomszédsági mártix
- Szomszédsági lista

	Létezik (u, v) él?	**Összes él listázása**	**Egy csúcs szomszédainak listázása**
Csúcsok + élek halmaza	$\Theta(\|E\|)$	$\Theta(\|E\|)$	$\Theta(\|E\|)$
Szomszédsági mátrix	$\Theta(1)$	$\Theta(\|V\|^2)$	$\Theta(\|V\|)$
Szomszédsági lista	$\Theta(\text{fokszám})$	$\Theta(\|E\|)$	$\Theta(\text{fokszám})$

Érdemes mindig elgondolkodni, hogy milyen reprezentációt választunk, az alapján, hogy milyen gráfogkra számítunk, azaz várhatóan milyen az élek és csúcsok eloszlása, azaz mennyire ritka / sűrű a gráf. Ha az élek száma arányos a csúcsok számával, az egy sűrű gráf, ha az élek száma arányos a csúcsok számának négyzetével, az egy ritka gráf.

Bonyolultságelmélet

---

1. Hatékony visszavezetés. Nemdeterminizmus. A P és NP osztályok. NP-teljes problémák.

A P osztály

R az eldönthető problémák osztálya.

Polinomidőben eldönthető problémák osztálya.

Tehát minden olyan **eldöntési probléma** P-ben van, amire létezik $O(n^k)$ időigényű algoritmus, valamely konstans k-ra.

> Ezeket a problémákat tartjuk **hatékonyan megoldhatónak**.

Elérhetőség

P-beli probléma.

Input: Egy $G = (V, E)$ irányított gráf. Feltehető, hogy $V = 1, \ldots, N$

Output: Vezet-e G-ben (irányított) út 1-ből N-be?

Erre van algoritmus:

- Kiindulásnak veszünk egy $X = 1$ és $Y = 1$ halmazt.
- Mindig kiveszünk egy elemet X-ből, és annak szomszédait betesszük X-be, és Y-ba is.
- Ez által $X \cup Y$-ban lesznek az 1-ből elérhető csúcsok.

Erre a konkrét implementációnk futásideje változó lehet, függhet például a gráf repretenzációtól, és a halmaz adatszerjezet megválasztásától. De a lényeg, hogy van-e polinom idejű algoritmus, és mivel általánosságban $O(N^3)$-el számolhatunk legrosszabb esetnek (előnytelen implementáció esetén is bele férünk), így $O(n^3)$-ös a futásideje az algoritmusnak (hiszen $N \leq n$, mert biztosan kevesebb a csúcsok száma, mint a gráfot ábrázoló biteké).

Hatékony visszavezetés

Rekurzív visszavezetés

> A.K.A. Turing-visszavezetés

Az A eldöntési probléma **rekurzívan visszavezethető** a B eldöntési problémára, jelben $A \leq_R B$, ha van olyan f **rekurzív függvény**, mely A inputjaiból B inputjait készíti **választartó** módon, azaz minden x inputra $A(x) = B(f(x))$

> Itt a *rekurzió* azt jelenti, hogy kiszámítható, adható rá algoritmus.

Ebben az esetben ha B eldönthető, akkor A is eldönthető, illetve ha A eldönthetetlen, akkor B is eldönthetetlen.

Lényegében ez azt fejezi ki, hogy "B legalább olyan nehéz, mint A".

Probléma ezzel a megközelítéssel: Ha A eldönthető probléma, B pedig nemtriviális, akkor $A \leq_R B$.

- Tehát nehézség szempontjából nem mondtunk valójában semmit.
- Ennek oka, hogy ebben az esetben az f inputkonvertáló függvényben van lehetőségünk egyszerűen az A probléma megoldására, és ennek megfelelően B egy *igen*, vagy *nem* példányának visszaadására.
- Ez alapján az összes nemtriviális probléma (azaz az olyanok, amik nem minden inputra ugyan azt adják) "ugyan olyan nehéznek" tűnik.
- Probléma oka: **Túl sok erőforrást engedünk meg az inputkonverzióhoz**, annyit, ami elég magának a problémának a megoldására.

- Megoldás: Hatékony visszavezetés.

Hatékony visszavezetés

> A.K.A. Polinomidejű visszavezetés

Az A eldöntési probléma **hatékonyan visszavezethető** a B eldöntési problémára, jelben $A \leq_P B$, ha van olyan f **polinomidőben kiszámítható** függvény, mely A inputjaiból B inputjait készíti **választartó** módon.

Ekkor ha B **polinomidőben** eldönthető, akkor A is eldönthető **polinomidőben**, illetve ha A-ra nincs polinomidejű algoritmus, akkor B-re sincs.

Példa

Egy példa a hatékony visszavezetésre a $PÁROSÍTÁS \leq SAT$

PÁROSÍTÁS

Input: Egy **CNF** (konjunktív normálformájú formula)

Output: Kielégíthető-e?

> Azaz van-e olyan értékadás, ami mellett igaz a formula?

SAT

Input: Egy G gráf

Output: Van-e G-ben teljes párosítás?

> Közös csúccsal nem rendelkező élek halmaza, amik lefednek minden csúcsot.

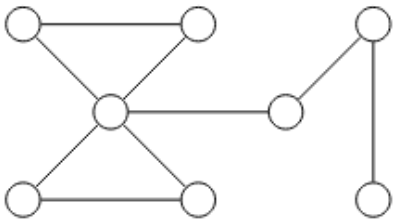
Visszavezetés

Tehát a cél egy G gráfból egy ϕ_G CNF előállítása választartó módon, polinomidőben úgy, hogy **G-ben pontosan akkor legyen teljes párosítás, ha ϕ_G kielégíthető**.

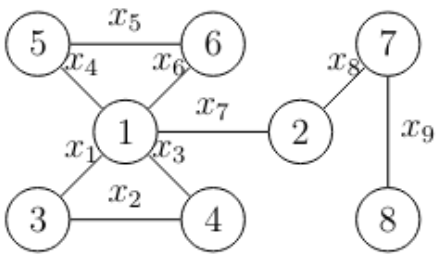
- Minden élhez rendelünk egy logikai változót.
- Akkor lesz igaz a változó, ha beválasztjuk az élt a teljes párosításba.
- A cél egy olyan CNF előállítása, amiben a következőt formalizáljuk: Minden csúcsra felítjuk, hogy pontosan egy él illeszkedik rá, majd ezeket összeéseljük. Ha így egy csúcsra sikerül megfelelő CNF-et alkotni, akkor azok összeéselése is CNF, hiszen CNF-ek éselése CNF.
- Egy csúcshoz annak formalizálása, hogy pontosan egy él fedi: legalább egy él fedi ÉS legfeljebb egy él fedi.
  - Legalább egy: Egyetlen CNF kell hozzá: $(x_1 \vee x_2 \vee \ldots \vee x_k)$.
  - Legfeljebb egy: Négyzetesen sok klóz kell hozzá, minden csúcspárra megkötjük, hogy "nem ez a kettő egyszerre": $\wedge\ 1 \leq i < j \leq k\ \neg(x_i \wedge x_j)$

$x_1, \ldots, x_k$ az adott viszgált csúcsra illeszkedő élek.

Nézzünk erre a fentire egy példát: ha a gráf már megint a



akkor felcímkézve a változókkal ezt kapjuk:



(adtunk a csúcsoknak is nevet közben)

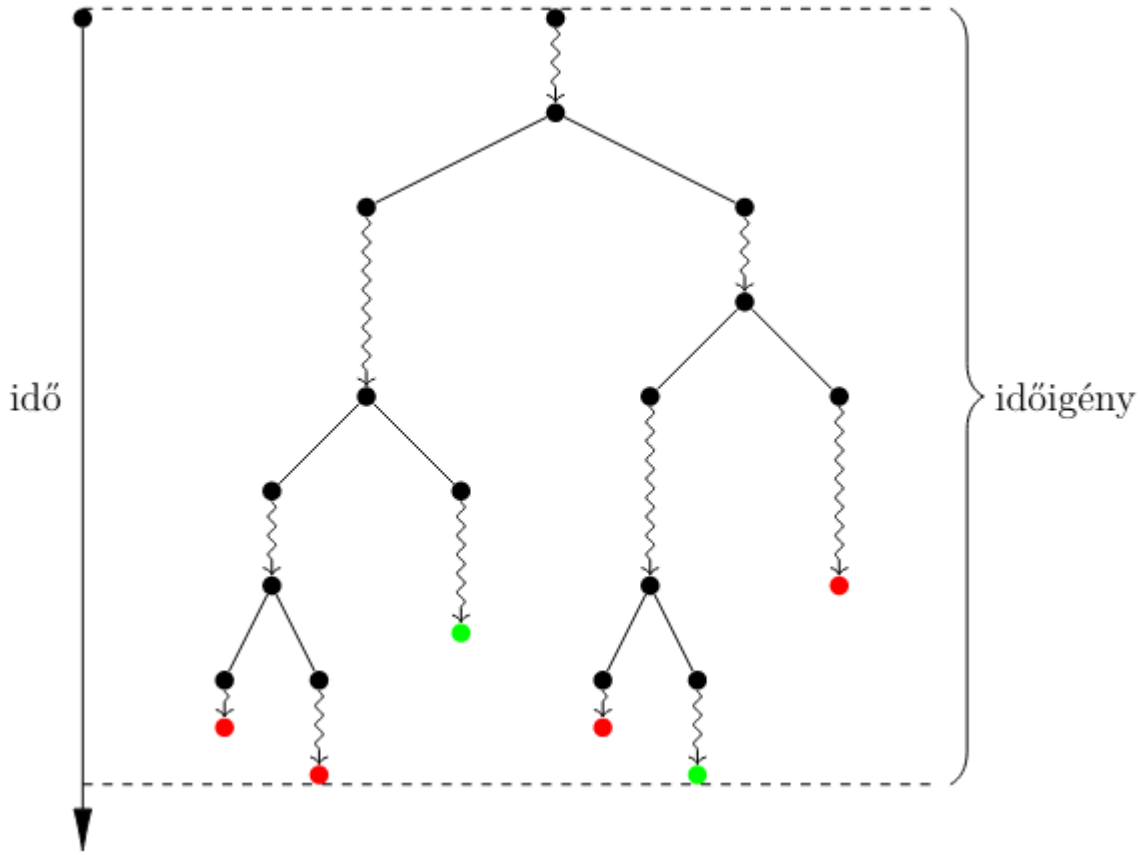
$$(x_4 \vee x_5) \wedge (\neg x_4 \vee \neg x_5) \wedge (x_5 \vee x_6) \wedge (\neg x_5 \vee \neg x_6) \wedge (x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$$
$$\wedge (x_2 \vee x_3) \wedge (\neg x_2 \vee \neg x_3) \wedge (x_7 \vee x_8) \wedge (\neg x_7 \vee \neg x_8) \wedge (x_8 \vee x_9) \wedge (\neg x_8 \vee \neg x_9)$$
$$\wedge x_9 \wedge (x_1 \vee x_3 \vee x_4 \vee x_6 \vee x_7) \wedge (\neg x_1 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_4) \wedge (\neg x_1 \vee \neg x_6) \wedge (\neg x_1 \vee \neg x_7)$$
$$\wedge (\neg x_3 \vee \neg x_4) \wedge (\neg x_3 \vee \neg x_6) \wedge (\neg x_3 \vee \neg x_7) \wedge (\neg x_4 \vee \neg x_6) \wedge (\neg x_4 \vee \neg x_7) \wedge (\neg x_6 \vee \neg x_7).$$

Nemdeterminizmus

Nemrealisztukus számítási modell: Nem tudjuk hatékonyan szimulálni.

RAM gépen el lehet képzelni a következő utasításképp: `v := nd()`.

Ezzel nemdeterminisztikusan adunk értéket egy bitnek, amit úgy lehet elképzelni, mintha ezen a ponton a számítás elágazna, és az egyik szálon `v = 1`, a másikon `v = 0` értékkel számol. Egy ilyen elégazásnak konstans időben kellene történnie.



A fenti képen egy **számítási fa** van, minden elégazás egy nemdeterminisztikus bitgenerálás.

Időigény: A leghosszabb szál időigénye. Tehát a **számítási fa mélysége**.

Eldöntési algoritmus esetén a végeredmény akkor **true, ha legalább egy szál true,** akkor **false, ha minden szál false**.

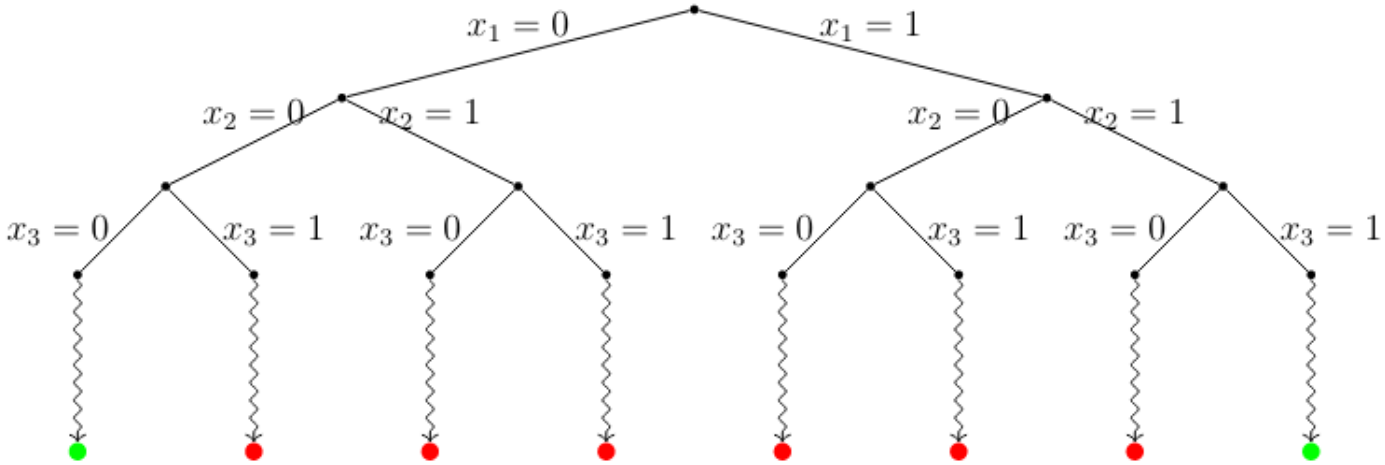
Nemdeterminisztikus algoritmus a SAT-ra

Input formulánkban az $x_1, \ldots, x_k$ változók fordulnak elő.

1. Generáljunk minden x_i-hez egy nemdeterminisztikus bitet, így kapunk egy értékadást.
2. Ha a generált értékadás kielégíti a formulát, adjunk vissza `true`-t, egyébként `false`-t.

Példa input: $(x_1 \vee \neg x_2) \wedge (x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_3)$

Ehhez az inputhoz a számítási fa:



Ennek az algoritmusnak a nemdeterminisztikus időigénye $O(n)$, hiszen n változónak adunk értéket, és a behelyettesítés, ellenőrzés is lineáris időigényű.

Az NP osztály

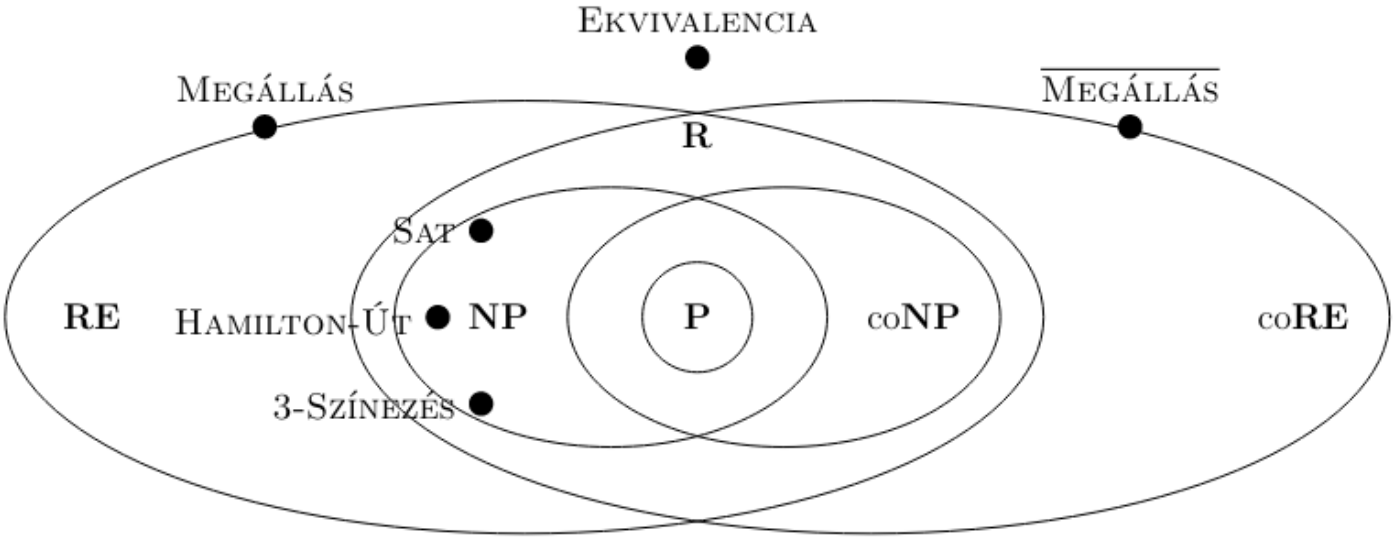
Nemdeterminisztikus algoritmussal polinomidőben eldönthető problémák osztálya.

A **SAT** a korábbi példa alapján például **NP-beli**.

$P \subset NP$ természetesen igaz, hiszen egy deterministikusan polinom idejű algoritmus felfogható olyan nemdeterminisztikusnak, ami sosem ágazik el. $P = coP$ miatt $P \subset NP \cap coNP$.

Ennél többet nem tudunk, nem tudjuk, hogy $P = NP$ igaz-e. Széleskörben elfogadott sejtés, hogy nem. Hasonlóan az sem ismert, hogy $NP = coNP$ igaz-e, erről is az az elfogadtott álláspont, hogy nem.

Persze $NP \subset R$ is igaz, mert a nemdeterminisztikus számítás szimulálható determinisztikusan, bár ez exponenciálisan lassú.



NP-teljes problémák

C**-teljesség definíciója**: Ha C problémák egy osztálya, akkor az A probléma

- C**-nehéz**, ha minden C-beli probléma visszavezethető A-ra
- C**-teljes**, ha A még ráadásul C-ben is van

Polinomidőben verifikálhatóság

Az A probléma polinomidőben verifikálható, ha van egy olyan R reláció, **inputok**, és **tanúk** között, melyre:

- Ha $R(I, T)$ az I inputra és a T tanúsítványra, akkor $|T| \leq |I|^c$ valamilyen c konstansra (azaz a tanúk "nem túl hosszúak")
- Ha kapunk egy (I, T) párt, arról determinisztikusan polinomidőben el tudjuk dönteni, hogy $R(I, T)$ fennáll-e, vagy sem (azaz egy tanú könnyen ellenőrizhető)
- Pontosan akkor létezik I-hez olyan T, melyre $R(I, T)$ igaz, ha I az A-nak egy "igen" példánya (azaz R tényleg egy jó "tanúsítvány-rendszer" az A problémához)

> SAT esetében pl. lineáris időben tudjuk ellenőrizni, hogy egy adott értékadás kielégíti-e a CNF-et.

Egy probléma pontosan akkor van NP-ben, ha polinomidőben verifikálható.

SAT

Cook tétele kimondja, hogy a **SAT egy NP-teljes probléma**.

Variánsok: FORMSAT, 3SAT is NP-teljes (és minden kSAT $k \geq 3$-ra), DE 2SAT P-beli, visszavezethető ugyanis az elérhetőségre.

Horn-átnevezhető formulák kielégítése is polinomidőben eldönthető.

> Horn-formula, ha minden klózban legfeljebb egy pozitív literál, Horn-átnevezhető, ha bizonyos változók komplementálásával Horn-formulává alakítható.

NP-teljes gráfelméleti problémák

Független csúcshalmaz

Input: Egy G irányítatlan gráf, és egy K szám

Output: Van-e G-ben K darab **független**, azaz páronként nem szomszédos csúcs?

Klikk

Input: Egy G gráf, és egy K szám.

Output: Van-e G-ben K darab páronként szomszédos csúcs?

Hamilton-út

Input: Egy G gráf.

Output: Van-e G-ben Hamilton-út?

Halmazelméleti NP-teljes problémák

Párosítás

Input: Két egyforma méretű halmaz, A, és B, és egy $R \subseteq A \times B$ reláció.

Output: Van-e olyan $M \subseteq R$ részhalmaza a megengedett pároknak, melyben minden $A \cup B$-beli elem pontosan egyszer van fedve?

> A halmaz: lányok, B halmaz: fiúk, reláció: ki hajlandó kivel táncolni. Kérdés: Párokba lehet-e osztani mindenkit?

Hármasítás

Input: Két egyforma méretű halmaz, A, B, és C, és egy $R \subseteq A \times B \times C$ reláció.

Output: Van-e olyan $M \subseteq R$ részhalmaza a megengedett pároknak, melyben minden $A \cup B \cup C$-beli elem pontosan egyszer van fedve?

> Hasonló példa áll, C halmaz házak, ahol táncolnak.

Pontos lefedés hármasokkal

Input: Egy U $3m$ elemű halmaz, és háromelemű részhalmazainak egy $S_1, \ldots, S_n \subseteq U$ rendszere.

Output: Van-e az S_i-k közt m, amiknek uniója U?

Halmazfedés

Input: Egy U halmaz, részhalmazainak egy $S_1, \ldots, S_n \subseteq U$ rendszere, és egy K szám.

Output: Van-e az S_i-k közt K darab, amiknek uniója U?

Halmazpakolás

Input: Egy U halmaz, részhalmazainak egy $S_1, \ldots, S_n \subseteq U$ rendszere, és egy K szám.

Output: Van-e az S_i-k közt K darab páronként diszjunkt?

Számelméleti NP-teljes problémák

Egész értékű programozás

Input: Egy $Ax \leq b$ egyenlőtlenség-rendszer, A-ban és b-ben egész számok szerepelnek.

Output: Van-e egész koordinátájú x vektor, mely kielégíti az egyenlőtlenségeket?

Részletösszeg

Input: Pozitív egészek egy $a_1, \ldots, a_k$ sorozata, és egy K célszám.

Output: Van-e ezeknek olyan részhalmaza, melynek összege épp K?

Partíció

Input: Pozitív egészek egy $a_1, \ldots, a_k$ sorozata.

Output: Van-e ezeknek egy olyan részhalmaza, melynek összege épp $\frac{\sum_{i=1}^{k} a_i}{2}$?

Hátizsák

Input: i darab tárgy, mindegyiknek egy w_i súlya, és egy c_i értéke, egy W összkapacitás és egy C célérték.

Output: Van-e a tárgyaknak olyan részhalmaza, melynek összsúlya legfeljebb W, összértéke pedig legalább C?

> TODO: erős-, gyenge NP-teljesség kell-e ide?

2. A PSPACE osztály. PSPACE-teljes problémák. Logaritmikus tárigényű visszavezetés. NL-teljes problémák.

A PSPACE osztály

Determinisztikusan (vagy nemdeterminisztikusan), polinomidőben megoldható problémák osztálya.

- $SPACE(f(n))$: Az $O(f(n))$ tárban eldönthető problémák osztálya.
- $NSPACE(f(n))$: Az $O(f(n))$ tárban **nemdeterminisztikusan** eldönthető problémák osztálya.
- $TIME(f(n))$: Az $O(f(n))$ időben eldönthető problémák osztálya.
- $NTIME(f(n))$: Az $O(f(n))$ időben **nemdeterminisztikusan** eldönthető problémák osztálya.

PSPACE-beli problémák még **nehezebbek, mint az NP-beliek**.

Fontos összefüggés NSPACE és SPACE között

$$NSPACE(f(n))\sube SPACE(f^2(n))$$

Ebből következik ez is:

$$PSPACE = NPSPACE$$

Hiszen a kettes hatványtól függetlenül $f(n)$ ugyan úgy csak egy **polinom**iális függvény.

> Ennek az összefüggésnek az oka, hogy a tár **újra felhasználható**. Emiatt viszonylag kevés tár is elég sok probléma eldöntésére. Az idő ezzel szemben sokkal problémásabb, nem tudjuk, hogy egy $f(n)$ időigényű nemdeterminisztikus algoritmust lehet-e $2^{O(f(n))}$-nél gyorsabban szimulálni.

Lineáris tárigény

Az előbb említett előny miatt elég sok probléma eldönthető $O(n)$ tárban.

Pl. **SAT**, **HAMILTON-ÚT,** és a **3-SZÍNEZÉS** mind eldönthető lineáris tárban. Csak lehetséges tanúkat kell generálni, fontos, hogy egyszerre csak egyet, ezt a tárat használjuk fel újra és újra. Ellenőrizzük a tanút, ha nem jó generáljuk a következőt.

Offline, vagy *lyukszalagos* tárigény

Ha az algoritmus az inputot csak olvassa, és az outputot *stream-mód*-ban írja, akkor az input, output regisztereket nem kell beszámolni, csak a working regisztereket.

> A cél ezzel az, mert a korábbiak alapján jó lenne, ha lehetne értelme szublineáris tárigénynek. Márpedig ha pl. az inputot már beszámoljuk, akkor az már legalább lineáris.

Az NL-osztály

- $L = SPACE(logn)$: Determinisztikusan logaritmikus tárban eldönthető problémák osztálya.
- $NL = NSPACE(logn)$: Nemdeterminisztikusan logaritmikus tárban eldönthető problémák osztálya.

Immermann-Szelepcsényi tétel szerint: $NL = coNL$

Mit nem szabad, hogy legyen esély NL-beli algoritmust készíteni?

- Az **inputot írni**.
- $\Theta(n)$ méretű bináris tömböt felvenni.

Mit szabad?

- Olyan **változót létrehozni**, amibe 0 és n közti számokat írunk, hiszen ezek $logn$ tárat igényelnek.
- Nem csak n-ig ér számolni, hanem bármilyen **fix fokszámú polinomig**. Pl. ha n^3-ig számolunk, az is elfér $log^3 = 3 * logn$ biten, tehát $O(logn)$ a tárkorlátja.
- Az **input valamelyik elemére rámutatni** egy pointerrel, hiszen lényegében ez is egy 0-tól n-ig értékeket felvevő változó.

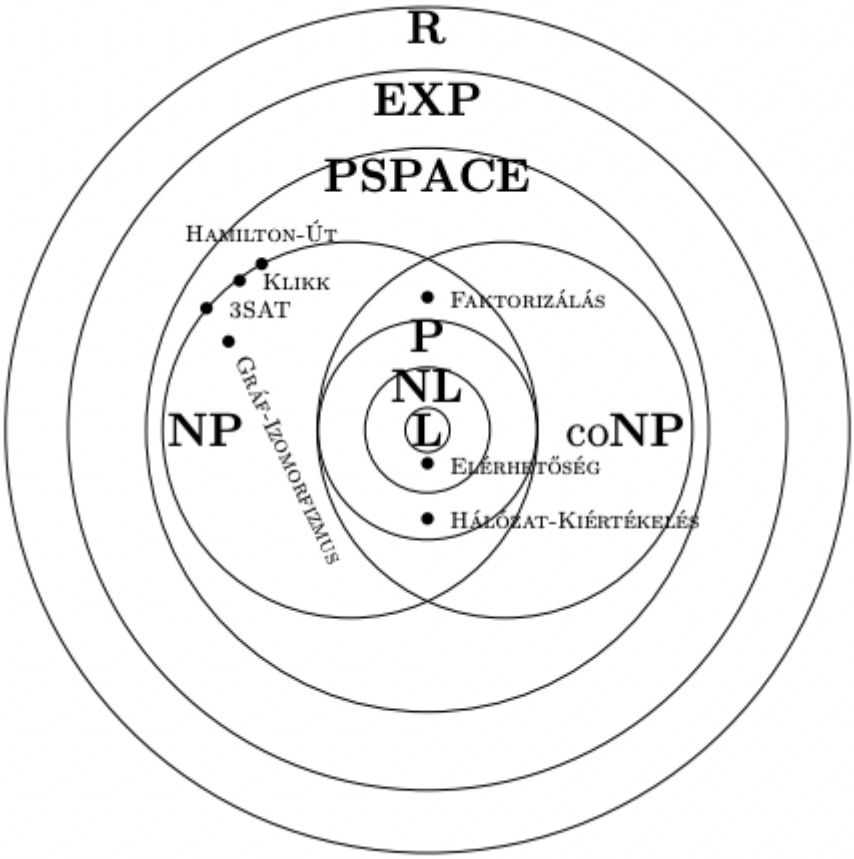
Elérhetőség

Determinisztikusan Savitch tétele szerint az ELÉRHETŐSÉG eldönthető $O(log^2 n)$ tárban. Ennek oka a rekurzió, hiszen egy példányunk $O(logn)$ táras, de ebből egyszerre akár $logn$ darab is lehet a memóriában.

Nemdeterminisztikusan bele férünk a logtárba. Ekkor "nemdeterminisztikusan bolyongunk" a gráfban, és ha N lépésben elértünk a csúcsig, akkor `true` amúgy `false`. Tehát minden iterációban átlépünk nemdeterminisztikusan minden szomszédra, ha megtaláltuk a cél csúcsot, `true`, ha nem tudunk már tovább lépni, vagy lefutott mind az N iteráció, akkor `false`.

Ezek alapján tehát:

$$\text{ELÉRHETŐSÉG} \in \mathbf{NL}$$



Logtáras visszavezetés

P-n belül ugye a polinomidejű visszavezetésnek nincs értelme. Hiszen ekkor az inputkonverziót végző függvényben meg tudjuk oldani a problémát, és csak visszaadni egy ismerten `true` vagy `false` inputot.

Definíció

Legyenek A és B eldöntési problémák. Ha f egy olyan függvény, mely

- A inputjaiból B inputjait készíti,
- választartó módon: A "igen" példányiból B "igen" példányait, "nem" példányaiból pedig "nem" példányt,
- és logaritmikus tárban kiszámítható,

akkor f egy logtáras visszavezetés A-ról B-re. Ha A és B közt létezik ilyen, akkor azt mondjuk, hogy A logtárban visszavezethető B-re, jelben $A \leq_L B$.

> f biztosan lyukszalagos, hiszen szublineárisnak kell lennie.

Tulajdonságok

A logaritmikus tárigényű algoritmusok polinom időben megállnak, hiszen $O(logn)$ tárat $2^{O(logn)}$ féleképp lehet teleírni, minden pillanatban a program K darab konstans utasítás egyikét hajtja éppen végre, így összesen $K * 2^{O(logn)}$-féle különböző konfigurációja lehet, ami polinom.

Ebből következik: $\text{Ha } A \leq_L B, \text{ akkor} A \leq_P B$

> Azaz a logtáras visszavezetés formailag "gyengébb".

Valójában nem tudjuk, hogy ténylegesen gyengébb-e ez a visszavezetés, de azt tudjuk, hogy akkor lesz gyengébb, ha $L \neq P$.

$L = P$ pontosan akkor teljesül, ha $\leq_L \quad = \quad \leq_P$

Ha f és g logtáras függvények, akkor kompozíciójuk is az. Ez azért jó, mert akkor itt is be lehet vetni azt a trükköt, amit a polinomidejű visszavezetésnél, azaz a C-nehézség bizonyításához elég egy már ismert C-nehéz problémát visszavezetni az adott problémára. Hiszen ekkor tranzitívan minden C-beli probléma visszavezethető lesz az aktuális problémára is.

NL-teljes problémák

Legyen $L \subset C \subseteq P$ problémák egy osztálya. Azt mondjuk, hogy az A probléma C-nehéz, ha C minden eleme **logtárban** visszavezethető A-ra.

Ha ezen kívül A még ráadásul C-beli is, akkor A egy C-teljes probléma.

> Szóval ugyan az, mint P-n kívül, csak logtárban, mivel P-n belül a polinomidejű visszavezetésnek nincs értelme.

P-teljes problémák

- Input egy **változómentes** ítéletkalkulus-beli formula, kiértékelhető-e?
- HÁLÓZAT-KIÉRTÉKELÉS

NL-teljes problémák

- **ELÉRHETŐSÉG**
- **ELÉRHETŐSÉG** úgy, hogy az input irányított, **körmentes** gráf
- **ELÉRHETETLENSÉG** (mivel ez az ELÉRHETŐSÉG komplementere, igy $coNL$-teljes, így NL -teljes, hiszen $NL = coNL$ az Immermann-Szelepcsényi tétel szerint)
- **2SAT** (, és annak a komplementere, megintcsak az Immermann-Szelepcsényi tétel miatt)

PSPACE-teljes problémák

QSAT

Input: Egy $\exists x_1 \forall x_2 \exists x_3 \ldots \forall x_{2m} \phi$ alakú **kvantifikált ítéletlogikai** formula, melynek magja, a ϕ konjunktív normálformájú, **kvantormentes** formula, melyben csak az $x_1, \ldots, x_{2m}$ változók fordulnak elő.

Output: Igaz-e ϕ?

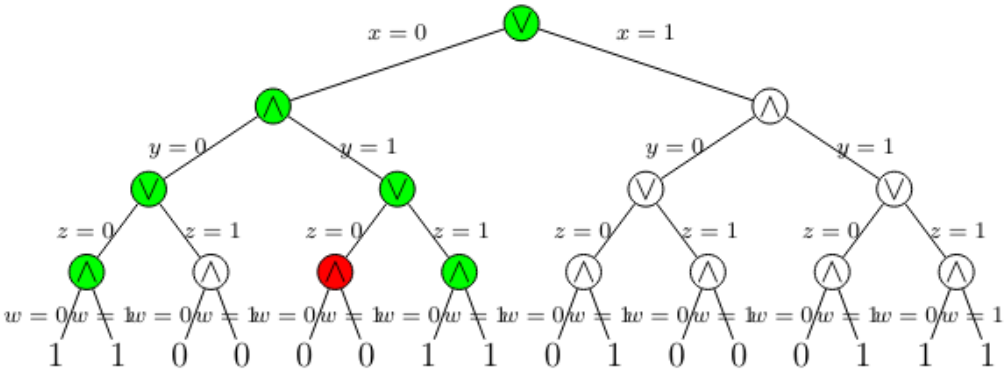
> Ez nem első rendű logika, az x_i változók csak igaz / hamis értékeket vehetnek fel.

A QSAT egy **PSPACE-teljes** probléma.

Egy QSAT-ot megoldó rekurzív algoritmus rekurziós fája:

Példa: a felső sorban látható formula rekurziós hívási gráfja lerajzolva:

$\exists x\ \forall y\ \exists z\ \forall w\big((\neg x \vee z \vee w)\ \wedge\ (y \vee \neg z)\ \wedge\ (x \vee \neg y \vee z)\big)$



Alul a 0/1-ek a formula magjának a kiértékelései az oda vezető értékadás mellett, a zöld csúcsok értéke 1, a pirosé 0, amelyik fehér maradt, azt (gyorsított kiértékelésnél) ki se kellett értékeljük, pl. mert a vagyolás bal oldala már 1 lett.

Tárigénye $O(n^2)$, mert a rekurziókor lemásoljuk az inputot, ami $O(n)$ méretű, és a mélység $O(n)$

QSAT, mint kétszemélyes, zéró összegű játék

Input: Egy $\exists x_1 \forall x_2 \exists x_3 \ldots \forall x_{2m} \phi$ alakú **kvantifikált ítéletlogikai** formula, melynek magja, a ϕ konjunktív normálformájú, **kvantormentes** formula, melyben csak az $x_1, \ldots, x_{2m}$ változók fordulnak elő.

Output: Az első játékosnak van-e nyerő stratégiája a következő játékban?

- A játékosok sorban értéket adnak a változóknak, előbb az első játékos x_1-nek, majd a második x_2-nek, megint az első stb., végül a második x_{2m}-nek.
- Ha a formula értéke igaz lesz, az első játékos nyert, ha hamis, a második.

FÖLDRAJZI JÁTÉK

Input: Egy $G = (V, E)$ irányított gráf. és egy kijelölt "kezdő" csúcsa.

Output: Az első játékosnak van-e nyerő stratégiája a következő játékban?

- Először az első játékos kezd, lerakja az egyetlej bábuját a gráf kezdőcsúcsára.
- Ezután a második játékos lép, majd az első, stb., felváltva, mindketten a bábut az aktuális pozíciójából egy olyan csúcsba kell húzzák, ami egy lépésben elérhető, és ahol még nem volt a játék során. Aki először nem tud lépni, vesztett.

További PSPACE-teljes problémák

- Adott egy M determinisztikus RAM program, és egy I inputja. Igaz-e, hogy M elfogadja I-t, méghozzá $O(n)$ tárat használva?
- Adott két reguláris kifejezés. Igaz-e hogy ugyan azokra a szavakra illeszkednek?

- Adott két nemdeterminisztikus automata. Ekvivalensek-e?
- $n \times n$-es SOKOBAN
- $n \times n$-es RUSH HOUR

Formális Nyelvek

1. Véges automata és változatai, a felismert nyelv definíciója. A reguláris nyelvtanok, a véges automaták, és a reguláris kifejezések ekvivalenciája. Reguláris nyelvekre vonatkozó pumpáló lemma, alkalmazása és következményei.

Véges automata

Az $M = (Q, \Sigma, \delta, q_0, F)$ rendszert **determinisztikus automatának** nevezzük, ahol:

- Q egy nem üres, véges halmaz, az **állapotok halmaza**
- Σ egy ábécé, au **input ábécé**
- $q_0 \in Q$ a **kezdő állapot**
- $F \subseteq Q$ a **végállapotok halmaza**
- $\delta : Q \times \Sigma \to Q$ egy leképezés, az **átmenetfüggvény**

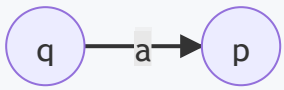
Példa:

- $Q = q_0, q_1, q_2$
- $\Sigma = a, b$
- $F = q_0$
- δ
  - $\delta(q_0, a) = q_1$
  - $\delta(q_1, a) = q_2$
  - $\delta(q_2, a) = q_0$
  - $\delta(q_0, b) = q_0$
  - $\delta(q_1, b) = q_1$
  - $\delta(q_2, b) = q_2$

Automata megadása irányított gráfként

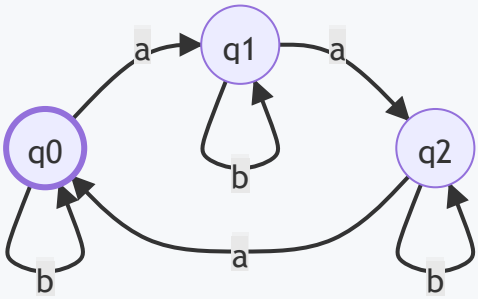
Gráf csúcsai az automata állapotai

Ha $\delta(q, a) = p$, akkor a q csúcsból egy élet irányítunk a p csúcsba, és az élet ellátjuk az a címkével



> Itt az automata a q állapotból az a input szimbólum hatására átmegy a p állapotba.

A korábbi példa automata megadása gráffal:



> A q_0 állapot jelen példában a végállapot is, amit a vastagított szél jelez.

Automata megadása táblázatként

Első sorban a kezdőállapot, végállapotokat meg kell jelölni (itt most csillag).

A korábbi példa automata megadása táblázattal:

δ	a	b
$^*q_0$	q_1	q_0
q_1	q_2	q_1
q_2	q_0	q_2

> Csillag jelzi, hogy az adott sor állapota végállapot.

Automata átmenetei

M konfigurációinak halmaza: $C = Q \times \Sigma^*$

A $(q, a_1 \dots a_n)$ **konfiguráció** azt jelenti, hogy M a q állapotban van és az $a_1 \dots a_n$ szót kapja inputként.

Átmeneti reláció

$(q, w), (q', w') \in C$ esetén $(q, w) \vdash_M (q', w')$, ha $w = aw'$, valamely $a \in \Sigma$-ra, és $\delta(q, a) = q'$.

> Azaz aminkor az automata átmegy q-ból q'-be, akkor az ehhez "felhasznált" szimbólumot leveszi az input szó elejéről. Pl. itt a hatására ment, és $w = aw'$, így az átmenet után az input szó már csak w' az a nélkül. Mondhatni, hogy az a-t felhasználta az átmenethez.

####### Átmeneti reláció fajtái

- $(q, w) \vdash_M (q', w')$: Egy lépés
- $(q, w) \vdash_M^n (q', w'), n \geq 0$: n lépés
- $(q, w) \vdash_M^+ (q', w')$: Legalább egy lépés
- $(q, w) \vdash_M^* (q', w')$: Valamennyi (esetleg 0) lépés

> Az M jelölés egy automatát azonosít, elhagyható, ha éppen csak 1 automatáról beszélünk, mert ilyenkor egyértelmű

> *, és + itt is, és mindenhol ebben a tárgyban úgy működik, mint megszokott regexeknél

Felismert nyelv

Az $M = (Q, \Sigma, \delta, q_0, F)$ automata által felismert nyelven az $L(M) = \{ w \in \Sigma^* \mid (q_0, w) \vdash_M^* (q, \epsilon) \text{ és } q \in F \}$ nyelvet értjük.

> Azaz q_0-ból w hatására valamelyik $q \in F$ végállapotba jutunk

> ϵ az üres szó

Nemdeterminisztikus automata

Az $M = (Q, \Sigma, \delta, q_0, F)$ rendszert **nemdeterminisztikus automatának** nevezzük, ahol:

- Q egy nem üres, véges halmaz, az **állapotok halmaza**
- Σ egy ábécé, az **input ábécé**
- $q_0 \in Q$ a **kezdő állapot**
- $F \subseteq Q$ a **végállapotok halmaza**
- $$\delta: Q \times \Sigma \to \mathcal{P}(Q)$$ egy leképezés, az **átmenetfüggvény***

> Azaz ugyan az, mint a determinisztikus, csak egy input szimbólum hatására egy állapotból többe is átmehet.

A determinisztikus automata ezen általánosítása (hiszen ez egy általánosítás, a determinisztikus automata is lényehében olyan nemdeterminisztikus ami mindig állapotoknak egy egyelemű halmazába tér át) **nem növeli meg a felismerő kapacitást**, tehát egy nyelv akkor és csak akkor ismerhető fel nemdeterminisztikus automatával, ha felismerhető determinisztikus automatával.

> Ezt "hatvány halmaz módszerrel" lehet bebizonyítani, meg kell nézni, hogy a hatására milyen állapotokba tud kerülni a nemdeterminisztikus automata, és azonkah az uniója lesz egy állapot. Ez a "determinizálás", aminek a során az állapotok száma nagyban megnőhet (akár exponenciálisan).

Átmeneti reláció

$(q, w), (q', w') \in C$ esetén $(q, w) \vdash_M (q', w')$, ha $w = aw'$, valamely $a \in \Sigma$-ra, és $q' \in \delta(q, a)$.

Felismert nyelv

Az $M = (Q, \Sigma, \delta, q_0, F)$ (nemdeterminisztikus) automata által felismert nyelven az $L(M) = \{ w \in \Sigma^* \mid (q_0, w) \vdash_M^* (q, \epsilon) \text{ valamely } q \in F\text{-re} \}$ nyelvet értjük.

> Azaz q_0-ból a w hatására elérhető valamely $q \in F$ végállapot. DE! Nem baj, ha elérhetően nem-végállapotok is.

Teljesen definiált automata

Akkor teljesen definiált egy automat, ha minden szót végig tud olvasni.

Azaz nem tud pl. egy $\delta(q, a) = \emptyset$ átmenet miatt elakadni.

Azaz akkor teljesen definiált, ha minden $q \in Q$ és $a \in \Sigma$ esetén $\delta(q, a)$ **legalább** egy elemű.

Determinisztikus automaták teljesen definiáltak, hiszen pontosan egy állapotba léphetünk tovább.

Nemdeterminisztikus automaták pedig teljesen definiálhatóvá tehetőek "csapda" állapot bevezetésével, anélkül, hogy a felismert nyelv megváltozna.

- Felveszünk egy q_c állapotot (ez a "csapda") állapot.
- $\delta(q, a) = \emptyset$ esetén legyen $\delta(q, a) = \{ q_c \}$
- Legyen $\delta(q_c, a) = \{ q_c \}$ minden $a \in \Sigma$-ra.

> A 3. pont az, ami miatt ez egy "csapda", nem lehet már ebből az állapotból kijönni.

Nemdeterminisztikus ϵ-automata

Tartalmaz ϵ-átmeneteket.

Az $M = (Q, \Sigma, \delta, q_0, F)$ rendszert **nemdeterminisztikus ϵ-automatának** nevezzük, ahol:

- Q egy nem üres, véges halmaz, az **állapotok halmaza**
- Σ egy ábécé, az **input ábécé**
- $q_0 \in Q$ a **kezdő állapot**
- $F \subseteq Q$ a **végállapotok halmaza**

- $\delta: Q \times (\Sigma \cup {\epsilon}) \to \mathcal{P}(Q)$ egy leképezés, az **átmenetfüggvény**

> Azaz ugyan olyan, mint a nemdeterminisztikus, csak lehet olyan átmenete, ami "nem fogyasztja" az inputot. Ez az ϵ-átmenet.

Ez sem bővíti a felismerő kapacitást, egy nyelv akkor és csak akkor ismerhető fel nemdeterminisztikus ϵ-átmenetes automatával, ha felismerhető nemdeterminisztikus automatával. ϵ automata ϵ-mentesítéssel átalakítható nemdeterminisztikus automatává, ekkor az automaza a q állapotból az a hatására azon állapotokba megy át, amelyekre M valamennyi (akár 0) ϵ-átmenettel, majd egy a-átmenettel jut el, továbbá az automata végállapotai azon az állapotok, amikből valamennyi (akár 0) ϵ-átmenettel egy F-beli állapotba jut.

Átmeneti reláció

$(q, w), (q', w') \in C$ esetén $(q, w) \vdash_M (q', w')$, ha $w = aw'$, valamely $a \in (\Sigma \cup {\epsilon}) - ra, és q' \in \delta(q, a)$.

> Ha $a = \epsilon$, akkor éppen $w = w'$

Felismert nyelv

Felismert nyelv definíciója ugyan az, mint a sima nemdeterminisztikus esetben.

Ekvivalencia tétel

Tetszőleges $L \subseteq \Sigma^*$ nyelv esetén a következő három állítás ekvivalens:

1. L reguláris (generálható reguláris nyelvtannal).
2. L felismerhető automatával.
3. L reprezentálható reguláris kifejezéssel.

Ezt külön három párra lehet belátni.

TODO: Ide reguláris nyelvtanokról, reguláris kifejezésekről röviden kéne, hiszen ezekről is szól a tétel

Reprezentálható nyelvek regulárisak

> $3 \to 1$ az ekvivalencia tételben.

Ha $L \subseteq \Sigma^*$ nyelv reprezentálható reguláris kifejezéssel, akkor generálható reguláris nyelvtannal.

Ez R struktúrája szerinti indukcióval belátható.

Reguláris nyelvek felismerhetők automatával

> $1 \to 2$ az ekvikalencia tételben.

Ha $L \subseteq \Sigma^*$ nyelv reguláris, akkor felismerhető automatával.

Ennek bizonyítását ez a két lemma képezi:

- Minden $G = (N, \Sigma, P, S)$ reguláris nyelvtanhoz megadható vele ekvivalens $G' = (N', \Sigma, P', S)$ reguláris nyelvtan, úgy, hogy P'-ben minden szabály $A \to B, A \to aB$, vagy $A \to \epsilon$ alakú, ahol $A, B \in N$ és $a \in \Sigma$.
- Minden olyan $G = (N, \Sigma, P, S)$ reguláris nyelvtanhoz, melynek csak $A \to B, A \to aB$ vagy $A \to \epsilon$ alakú szabályai vannak, megadható olyan $M = (Q, \Sigma, \delta, q_0, F)$ nemdeterminisztikus ϵ-automata, amelyre $L(M) = L(G)$.