Funkcionális programozás
12. előadás

MÁRTON Gyöngyvér

Sapientia Egyetem, Matematika-Informatika Tanszék
Marosvásárhely, Románia
mgyongyi@ms.sapientia.ro

2025, tavaszi félév

Miről volt szó?

- típusok és adatszerkezetek: rekord típusok
- ByteString-ek
- algebrai adattípusok

Miről lesz szó?

- kombinatorikai feladatok
  - lexicografikus sorrend,
  - az `n` elem `m`-ed rendű kombinációi,
  - az `n` királynő feladat
  - adott összeg előállítása
  - részhalmazok előállítása
  - Pascal-háromszög
- bináris keresés (list, array típus): névnapok meghatározása

Kombinatorikai feladatok

1. feladat

Írjunk egy Haskell-függvényt, amely meghatározza az összes olyan `m` hosszúságú listát, amely egy bemeneti lista elemeiből képezhető.

- az `lGen_` halmazműveleteket alkalmaz, létrehozza azokat a listákat, amelyek első elemei rendre a bemeneti lista elemei lesznek, a többi elemet pedig rekurzívan generálja
- a rekurzív hívásban a függvény első paramétere az eredeti lista lesz, a második paramétert pedig minden egyes alkalommal csökkentjük eggyel

```
lGen_ :: (Eq a) => [a] -> Int -> [[a]]
lGen_ ls 0 = [[]]
lGen_ ls m = [k : ve | k <- ls, ve <- lGen_ ls (m-1)]

> lGen_ [0, 1] 4
[[0,0,0,0],[0,0,0,1],[0,0,1,0],[0,0,1,1],[0,1,0,0],[0,1,0,1],[0,1,1,0],
[0,1,1,1],[1,0,0,0],[1,0,0,1],[1,0,1,0],[1,0,1,1],[1,1,0,0],[1,1,0,1],
[1,1,1,0],[1,1,1,1]]

> lGen_ "ab" 3
["aaa","aab","aba","abb","baa","bab","bba","bbb"]
```

Kombinatorikai feladatok

- a következő `lGen`-ben módosítjuk a listaelemek generálási módjának a sorrendjét
- először generáljuk a `ve` listaelemeket, és csak ezután adjuk meg a `k` értékének a generálási módját:

```
lGen :: (Eq a) => [a] -> Int -> [[a]]
lGen ls 0 = [[]]
lGen ls m = [k : ve | ve <- lGen ls (m-1), k <- ls]
```

- a módosítás a listaelemek más sorrendjét fogja eredményezni:

```
> lGen "ab" 3
["aaa","baa","aba","bba","aab","bab","abb","bbb"]
```

- az algoritmus hatékonyságát is javítottuk, amelyet a következő oldalon megadott `foGen` és a `foGen_` időigényei közötti lényeges különbség jelez

Kombinatorikai feladatok

```
foGen :: (Show a, Eq a) => [a] -> Int -> IO ()
foGen ls m = do
    let rLs = lGen ls m
    print $ last rLs

> :set +s
> foGen [0,1] 20
[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]
(0.67 secs, 352,405,392 bytes)

foGen_ :: (Show a, Eq a) => [a] -> Int -> IO ()
foGen_ ls m = do
    let rLs = lGen_ ls m
    print $ last rLs

> foGen_ [0,1] 20
[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]
(3.93 secs, 3,120,641,320 bytes)
```

- a `foGen`, illetve `foGen_` csak a kigenerált lista utolsó elemét írja ki, így a függvények időigényének értékét nem befolyásolta a listaelemek kiíratásának időigénye

n elem m-ed rendű kombinációi

2. feladat

Írjunk egy Haskell-függvényt, amely meghatározza n *elem* m*-ed rendű kombinációit.*

```
komb :: (Ord a) => [a] -> Int -> [[a]]
komb ls 0 = [[]]
komb ls m = [k : ve | ve <- komb ls (m-1),
                      k <- ls, feltKomb k ve]

feltKomb x [] = True
feltKomb x (k : ve)
    | k <= x = False
    | otherwise = feltKomb x ve

> komb [4, 1, 7, 9] 3
[[1,4,7],[1,4,9],[4,7,9],[1,7,9]]

> komb "wxyz" 2
["wx","wy","xy","wz","xz","yz"]
```

n elem m-ed rendű kombinációi

- a `komb` az `lGen`-en alapszik
- a `komb` paraméterként egy listát és az `m` értéket kapja meg, ahol a lista elemszáma adja a feladat megfogalmazásában szereplő `n` értéket
- mivel a kigenerált listák nem mindegyike felel meg a feladat kritériumának, ezért egy tesztelő `feltKomb` függvényt is alkalmazunk
- a `feltKomb` kimenete akkor lesz `True` ha a bemeneti lista elemei szigorúan növekvő sorrendben vannak
- például a fenti bemenet esetében az eredmény lista nem tartalmazhatja az `[1, 4, 7]`, `[1, 7, 4]`, `[7, 4, 1]` stb. mindegyikét, csak az `[1, 4, 7]`-t fogja tartalmazni
- a `feltKomb`-ban, ha módosítjuk a feltételt `k >= x = False`-ra, akkor a `[7, 4, 1]` fog szerepelni az eredménylistában
- a következő kódsor a `feltKomb` egy kompaktabb változata:

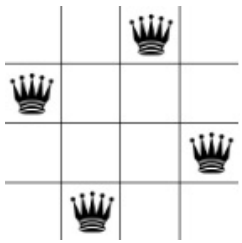
```
feltKomb_ :: (Ord a) => a -> [a] -> Bool
feltKomb_ x ls = all (\k -> x < k) ls
```

Az n királynő feladat

3. feladat

Írjunk egy Haskell-függvényt, amely elhelyez egy sakktáblán 8 királynőt úgy, hogy azok ne üssék egymást. Általánosan is oldjuk meg a feladatot, azaz helyezzünk el egy nxn-es sakktáblán n királynőt úgy, hogy azok ne üssék egymást.

- a gyakorlatban egy `nxn`-es sakktáblán, ha el akarunk helyezni `n` királynőt úgy, hogy azok ne üssék egymást, ez azt fogja jelenteni, hogy nem tehetjük őket ugyanabba a sorba, oszlopba, illetve átlóra
- a megoldás listák kigenerálását fogja jelenti:
  - a listaelemek egész számok lesznek
  - az első listaelem az első sorban levő királynő oszlop-pozícióját fogja jelenteni, a második listaelem a második sorban levő királynő oszlop-pozícióját jelenti, és így tovább
- egy `4x4`-es sakktáblán `[3, 1, 4, 2]` helyes megoldás azt fogja jelenteni, hogy a királynőt az első sorban a harmadik, a második sorban az első, a harmadik sorban a negyedik és a negyedik sorban a második oszlopba tettük:



Az n királynő feladat

```
kiralyno :: Int -> [[Int]]
kiralyno n = auxK n n
    where
    auxK :: Int -> Int -> [[Int]]
    auxK n 0 = [[]]
    auxK n m = [k : ve |
            ve <- auxK n (m-1),
            k <- [1..n], feltKir k ve ]
feltKir :: Int -> [Int] -> Bool
feltKir x ls = auxFeltK 1 x ls
    where
    auxFeltK :: Int -> Int -> [Int] -> Bool
    auxFeltK i x [] = True
    auxFeltK i x (k : ve)
        | k == x = False
        | abs (x - k) == i = False
        | otherwise = auxFeltK (i + 1) x ve

> kiralyno 5
[[4,2,5,3,1],[3,5,2,4,1],[5,3,1,4,2],[4,1,3,5,2],[5,2,4,1,3],
[1,4,2,5,3],[2,5,3,1,4],[1,3,5,2,4],[3,1,4,2,5],[2,4,1,3,5]]
```

Az `n` királynő feladat

- az algoritmus a kombinációkat meghatározó függvény egy változata lesz
- a feltételek teljesülését vizsgáló függvény a `feltKir` lesz
- ennél a feladatnál a listaelemek nem kell növekvő sorrendben legyenek, elég ha különböznek
- az eredménylisták elemeit akkor határozzuk meg, amikor *jövünk vissza* a rekurzióból, például az `5x5`-ös sakktábla esetében a `[4,2,5,3,1]` lista elsőként meghatározott eleme az `1` lesz, majd eléje kerül a `3`, majd az elé az `5`-ös, és így tovább
- az átlók menti ütközés kizárását a `abs (x - k) == i = False` feltétel biztosítja
- ha igaz a fenti feltétel az azt jelenti, hogy az `x` elem a már kigenerált listabeli elemek közül a `k`-val, amely `i` pozíciónyira helyezkedik el az `x`-től, az átlók mentén ütközni fog,
- például a `6x6`-os sakktábla esetében a `[2,6,3]` lista elejére nem lehet `1`-est tenni, mert az átlós ütközést eredményezne a `2`-vel (`abs (2 - 1) == 1`), de `4`-est sem lehet tenni, mert az a `6`-tal okozna átlós ütközést (`abs(6 - 4) == 2`)

Az n királynő feladat

a `feltKir` tömörebb változata a következő:

```
feltKir_ :: Int -> [Int] -> Bool
feltKir_ x ls = auxFeltK x ls
    where
    auxFeltK :: Int -> [Int] -> Bool
    auxFeltK x ls = all (aux x) $ zip [1..] ls
        where
        aux x (i, k) = (k /= x) && (abs(x - k) /= i)
```

Az n királynő feladat

A feladat megoldásait elegánsabban is meg tudjuk jeleníteni a képernyőn, a `foKiralyno` meghívásával:

```
kiirSor :: Int -> Int -> IO()
kiirSor n k = do
    mapM_ (auxF k) [1..n]
    putStrLn ""
        where
        auxF k x =
            if k == x then putStr "Q " else putStr ". "

kiirTabla :: Int -> [Int] -> IO()
kiirTabla n ls = do
    mapM_ (kiirSor n) ls
    putStrLn ""

foKiralyno :: Int -> IO()
foKiralyno n = mapM_ (kiirTabla n) $ kiralyno n

> foKiralyno 5
```

Kombinatorikai feladatok

- észrevehető az algoritmikai hasonlóság az `lGen`, `komb`, illetve `kiralyno` függvények között
- ezek megadhatóak általánosan: különböző feltételeket definiáló függvényt írunk a megfelelő listák kiszűrésére, a generálást pedig a `rekurzio` fogja végezni:

```
rekurzio :: Int -> Int -> (Int -> [Int] -> Bool) -> [[Int]]
rekurzio n 0 fg = [[]]
rekurzio n m fg = [k : ve | ve <- rekurzio n (m-1) fg, k <- [1..n], fg k ve]

lGenR :: Int -> Int -> [[Int]]
lGenR n m = rekurzio n m (\k ve -> True)

> lGenR 2 3
[[1,1,1],[2,1,1],[1,2,1],[2,2,1],[1,1,2],[2,1,2],[1,2,2],[2,2,2]]

permutacioR :: Int -> [[Int]]
--permutacioR n = rekurzio n n (\k ve -> notElem k ve)
permutacioR n = rekurzio n n notElem

> permutacioR 3
[[3,2,1],[2,3,1],[3,1,2],[1,3,2],[2,1,3],[1,2,3]]
```

Kombinatorikai feladatok

```
feltKomb :: (Ord a) => a -> [a] -> Bool
feltKomb x = all (aux x)
    where
    aux x k = k > x
```

```
kombR :: Int -> Int -> [[Int]]
kombR n m = rekurzio n m feltKomb
```

```
> kombR 3 2
[[1,2],[1,3],[2,3]]
```

```
feltKir :: Int -> [Int] -> Bool
feltKir x ls = auxFeltK 1 x ls
    where
    auxFeltK :: Int -> Int -> [Int] -> Bool
    auxFeltK i x ls = all (aux x) $ zip [1..] ls
        where
        aux x (i, k) = k /= x) && (abs(x - k) /= i)
```

```
kiralynoR :: Int -> [[Int]]
kiralynoR n = rekurzio n n feltKir
```

```
> kiralynoR 6
[[5,3,1,6,4,2],[4,1,5,2,6,3],[3,6,2,5,1,4],[2,4,6,1,3,5]]
```

Adott összeg előállítása

4. feladat

Írjunk egy Haskell-függvényt, amely meghatározza, hogy hányféleképpen állítható elő egy adott `sumV` összeg az `ls` listában megadott számokból, ha mindegyik számot csak egyszer használhatjuk fel.

- a kombinációkat kigeneráló (`komb`) függvényt fogjuk használni, mert a feladat átfogalmazható: határozzuk meg az adott `ls` listából képezhető `1, 2, ..., m` elemű kombinációkat úgy, hogy az elemek összege egyenlő legyen egy adott `sumV` értékkel

A feladat megoldható hatékonyabban:

- meghatározzuk az `ls` listából előállítható részhalmazokat, kivéve az üres halmazt,
- a részhalmazokból válogatunk, a feltételnek megfelelően: kiválasztjuk azokat, amelyek elemeinek összege `sumV`

Adott összeg előállítása

```
felSum1 :: Int -> [Int] -> [[Int]]
felSum1 sumV ls = auxSum 1 (length ls) sumV ls
    where
    auxSum :: Int -> Int -> Int -> [Int] -> [[Int]]
    auxSum i len s ls
        | i == len = []
        | otherwise =
            [ kLs | kLs <- komb ls i, sum kLs == s]
            ++ auxSum (i+1) len s ls

> felSum1 13 [4, 2, 7, 9]
[[4,9],[2,4,7]]
```

- az auxSum az 1, 2, ..., length ls rendű kombinációk közül válogat, és a ++ operátorral egy kimeneti listába fűzi a feltételnek eleget tevő listákat

Részhalmazok előállítása

A feladat hatékonyabb megoldása:

- meghatározzuk az `ls` listából előállítható részhalmazokat, kivéve az üres halmazt,
- a részhalmazokból válogatunk, a feltételnek megfelelően: kiválasztjuk azokat, amelyek elemeinek összege `sumV`

```
felSum2 :: Int -> [Int] -> [[Int]]
felSum2 sumV ls = [ kLs | kLs <- reszH ls, sum kLs == sumV ]

reszH :: [Int] -> [[Int]]
reszH [] = []
reszH (k : ve) = auxH k nVe ++ nVe
    where
    nVe = reszH ve

auxH :: Int -> [[Int]] -> [[Int]]
auxH x [] = [[x]]
auxH x (kL : veL) = (x : kL) : auxH x veL

> felSum2 30 [1, 2, 3, 5, 7, 8, 9, 10, 15]
[[1,2,3,5,9,10],[1,2,3,7,8,9],[1,2,3,9,15],[1,2,5,7,15],
...
```

Részhalmazok előállítása

```
> reszH [1,2,3]
[[1,2,3],[1,2],[1,3],[1],[2,3],[2],[3]]

> auxH 3 [[2, 1], [2], [1], []]
[[3,2,1],[3,2],[3,1],[3]]
```

A `reszH` az `1,2,3,4` elemekből a következőképpen állítja elő a megfelelő részhalmazokat:

- a kiindulási pont az üres lista lesz, ezt fogjuk rendre bővíteni az `1, 2, 3, 4, ...` elemekkel
- feltételezzük, hogy előállítottuk az `1,2` elemekből képezhető részhalmazokat:

  `[[2,1],[2],[1],[]]`

- az `1,2,3` elemekből képezhető részhalmazokat az előző lista segítségével határozzuk meg:
  - beszúrjuk a 3-as elemet minden már előállított részhalmazba (ezt végzi az `auxH` függvény):

    `[[3,2,1],[3,2],[3,1],[3]]`

  - egymásután fűzzük a két listát:

    `[[3,2,1],[3,2],[3,1],[3],[2,1],[2],[1],[]]`

A Pascal-háromszög

5. feladat

Írjunk egy Haskell függvényt, amely kiírja a Pascal háromszög első `n` sorát egy szövegállományba, háromszög formában.

- a Pascal-háromszög a binomiális együtthatók háromszög formában való rendezését jelenti
- a binomiális együttható azt a pozitív egész számot jelenti, amely az `(1 + x)`$^{\texttt{n}}$ polinom `x`$^{\texttt{k}}$ tagjának az együtthatója, amely ugyanakkor egyenlő lesz `n` elem `k`-ad rendű kombinációinak a számával
- a Pascal-háromszög kiíratása során az `n` értéke a sor, míg a `k` értéke az oszlop értékét jelöli, és fennáll: $\texttt{n} \geq \texttt{k} \geq 0$
- a binomiális együtthatók értékét direkt módon a következő képlettel is meg tudjuk határozni:

$$\binom{\texttt{n}}{\texttt{k}} = \frac{\texttt{n!}}{\texttt{k!} \cdot \texttt{(n-k)!}}$$

- a binomiális együtthatók értékét azonban meg lehet határozni úgy is, hogy az `(n-1)`-edik sorban levő értékekből generáljuk az `n`-edik sorban levő értékeket, ahol a nulladik sorban az `1` érték található, amely $\binom{0}{0}$ értéknek felel meg

A Pascal-háromszög

- a `pascal` függvény az előző oldal utolsó pontjában megadott a gondolatmenetet követi
- a nulladik sort egy egyelemű listaként, az 1-es értékkel definiálja
- a következőt, azaz az első sort a nulladik, a második sort az első stb. alapján fogja meghatározni
- az új sor első elemként mindig az 1-es értéket rögzíti, majd az előző sor elemei alapján a többi elemet az `auxPascal` függvénnyel határozza meg:
  - az előző sorban (listában) levő egymás melletti két elemet összeadja, majd a kapott értékekből felépíti az új listát, azaz az új sor elemeit
  - az előző sor (lista) utolsó elemét pedig változatlanul teszi át az új listába
- példa, az első négy sor meghatározására:

```
[1] → 1 :[1] → [1,1]
[1,1] → 1 : [1+1, 1] → [1,2,1]
[1,2,1] → 1 : [1+2, 2+1, 1] → [1,3,3,1]
[1,3,3,1] → 1 : [1+3, 3+3, 3+1, 1] → [1,4,6,4,1]
```

A Pascal-háromszög

```
auxPascal :: Integral a => [a] -> [a]
auxPascal [k] = [k]
auxPascal (k1 : k2 : ve) = (k1 + k2) : auxPascal (k2 : ve)

pascal :: Integral a => Int -> [a]
pascal 0 = [1]
pascal n = 1 : auxPascal (pascal (n-1))

> pascal 5
[1,5,10,10,5,1]

> auxPascal [1, 5, 10, 10, 5, 1]
[6,15,20,15,6,1]
```

A Pascal-háromszög

- a Pascal-háromszög első `n` sorának a generálását a `pascalN` fogja végezni, ez a korábbi `pascal` függvény módosított változata lesz
- a generált listákat egymás után fűzzük a `++` operátorral, az `auxPascal` függvény bemenetének pedig az eredménylistához utolsónak hozzáfűzött listát adjuk meg

```
pascalN :: Integral a => Int -> [[a]]
pascalN 0 = [[1]]
pascalN n = temp ++ [1 : auxPascal (last temp)]
    where
    temp = pascalN (n - 1)

> pascalN 4
[[1],[1,1],[1,2,1],[1,3,3,1],[1,4,6,4,1]]
```

A Pascal-háromszög

- ahhoz, hogy háromszög formában tudjuk megjeleníteni a számokat, a számok közé egy-egy szóközt, illetve minden egyes sor elé, bizonyos számú szóközt kell tenni
- a sorok elejére kerülő szóközök számát a `db` változó jelöli, és a `szokozok` függvényben kerül pontos meghatározásra
- a `db` érték attól függ, hogy hány számjegy, illetve szóköz található a Pascal-háromszög legutolsó, illetve aktuális sorában

```
szokozok :: String -> Int -> String
szokozok ls i = replicate db ' '
    where
    db = div (i - length ls) 2
```

A Pascal-háromszög

- a Pascal-háromszög legutolsó sorában található szóközök és számjegyek számát az `i` jelöli, és a `printPascal`-ban, a feladat *főfüggvényében* kerül meghatározásra
- a `listaToStr` függvény első paramétere ez az `i` érték lesz, második paramétere pedig a Pascal-háromszög soraiban levő számok lesznek
- a `listaToStr` az `auxSor`-t alkalmazva, betesz az aktuális sor elejére `db` darab szóközt, majd az aktuális sorban levő további számokat az `auxSz`-et meghívva, szóközökkel elválasztva összefűzi

```
listaToStr :: Show a => Int -> [[a]] -> [String]
listaToStr i = map (auxF i)

auxF :: Show a => Int -> [a] -> String
auxF i k = auxSor i k ++  auxSz k

auxSor :: Show a => Int -> [a] -> String
auxSor i k = szokozok (unwords $ map show k) i

auxSz :: Show a => [a] -> String
auxSz [] = ""
auxSz (k : ve)= show k ++ " " ++ auxSz ve
```

A Pascal-háromszög

Az állományba való kiíratást a `printPascal` végzi, ez lesz a feladat főfüggvénye, amelynek paraméterként meg kell adni az állomány nevét, illetve hogy hány sort kell meghatározni a Pascal-háromszögből:

```
printPascal :: String -> Int -> IO ()
printPascal nev n = do
    let pLs = pascalN n
    let i = length $ unwords $ map show $ last pLs
    let tLs = listaToStr i pLs
    outf <- openFile nev WriteMode
    mapM_ (hPutStrLn outf) tLs
    hClose outf

 > printPascal "pascal30.txt" 30
```

Bináris keresés - névnapok meghatározása

6. feladat

Írjunk egy Haskell-programot, amely a `Datum` és `Személy` adatszerkezeteket használva megállapítja egy adott személyről, hogy mikor van a névnapja.

```
data Datum = Datum {
   dNap :: Int,
   dHonap:: Int,
   dEv :: Int
   } deriving (Show)

data Szemely = Szemely {
   szVnev :: [Char],
   szKnev :: [Char],
   szSzulD :: Datum
   } deriving (Show)
```

A névnapok megállapításához használjuk a `nevnapok.txt` állományt, amely letölthető a következő linkről: *https://ms.sapientia.ro/~mgyongyi/Funk_Log/Jegyzet/nevnapok.txt*

Bináris keresés - névnapok meghatározása

- a `nevnapok.txt`-ben ábécésorrendben találhatók a különböző nevek: minden egyes sorban egy név szerepel, és a név után zárójelben megjelennek a névnapok
- a nevek írásakor nincsenek ékezetes betűk használva
- a különböző betűvel kezdődő névnapok között üres sorok vannak, illetve az azonos kezdőbetűvel kezdődő nevek előtt fel van tüntetve a kezdőbetű
- a feladat főfüggvénye a `foNevnap` lesz, amely az `str`-be kiolvassa a `hGetContents` segítségével a `nevnapok.txt` teljes tartalmát
- az `str` feldarabolását a `splitOn` végzi, ehhez importálni kellett a `Data.List.Split`-et

```
> splitOn "0" "120123012340 12"
["12","123","1234","12"]

> splitOn "00" "11001001111010011111 01"
["11","1","111101","1111101"]
```

- a `splitOn` kimenete alapján létrehozzunk egy tömböt, amelyet az `ar` fog jelölni
- a `foNevnap` az `ar` tömbben bináris kereséssel határozza meg egy adott személy névnapját, ahol a keresett személy adatait konstans értékként adjuk meg

Bináris keresés - névnapok meghatározása

```
import Data.Array ( Array, (!), listArray )
import Data.List.Split ( splitOn )
import Control.Exception ( SomeException, catch )
import System.IO

foNevnap :: IO ()
foNevnap =
   catch ( do
       inf <- openFile "nevnapok.txt" ReadMode
       str <- hGetContents inf
       let ls = splitOn "\n" str
       let m = length ls
       let ar = listArray (0, m-1) ls
       let egySz = Szemely "Kiss" "Szabolcs"
                   (Datum 10 5 1945)
       let res = bKeresA1 compA ar (szKnev egySz) 0 (m-1)
       if res == -1 then print "Nincs benne!!"
       else print $ ar ! res
       hClose inf
       ) hibaKezelo
       where
       hibaKezelo :: SomeException -> IO ()
       hibaKezelo err = putStrLn $ "IO hiba!!: "
                        ++ show err

> foNevnap
"Szabolcs (julius 17., julius 28., szeptember 19.)"
```

Bináris keresés - névnapok meghatározása

- a `bKeresA1` bináris keresést alkalmaz az `ar` tömbben
- az elemek összehasonlítását az első paramétereként megadott `fg` függvény szerint végzi, amely a `bKeresA1` meghívásakor a `compA` függvényértéket fogja felvenni
- találat esetén a függvény kimenete a keresett elem tömbbeli pozíciója lesz, ellenkező esetben pedig `-1` lesz

```
bKeresA1 :: ([Char] -> [Char] -> Ordering) ->
            Array Int [Char] -> [Char] -> Int -> Int -> Int
bKeresA1 fg a k l h
    | h < l = -1
    | cRes == LT = bKeresA1 fg a k l (i - 1)
    | cRes == GT = bKeresA1 fg a k (i + 1) h
    | cRes == EQ = i
      where
      cRes = fg k aElem
      i = div (l + h) 2
      aElem = a ! i
```

Bináris keresés - névnapok meghatározása

- a `compA` a `compare` beépített függvény segítségével oldja meg a nevek, azaz a `String` típusú értékek összehasonlítását
- a `compA` második paramétere az `ar` tömb egy eleme lesz, amelynek szerkezete a következő: `Abbas (november 12.)`
- a `compA`-ban a `takeWhile`-t alkalmaztuk, hogy a tulajdonképpeni nevet meghatározhassuk, azaz lekértük a karaktereket az első szóközig

```
compA :: [Char] -> [Char] -> Ordering
compA s1 s2 = compare s1 nS2
        where
        nS2 = takeWhile (/= ' ') s2
```