

## 1. Listák használata: rendezés

Néhány rendezőalgoritmus SML-ben: beszűró rendezés (inssort), gyorsrendezés (quicksort), összetettsű rendezés (tmsort és bmsort), simarendezés (simsort).

### 1.1. Beszűró rendezés

Az ins segédfüggvény az xs elemet a megfelelő helyre rakja be az ys listában:

```
(* ins (x, ys) = az x értékkel a <= reláció szerint
   bővíttett ys
   ins : real * real list -> real list
   PRE: ys a <= reláció szerint rendezett
*)
fun ins (x, y:ys) =
  if x <= y then x::y::ys else y::ins(x, ys)
| ins (x : real, []) = [x]
```

inssort-tal rekurzívan rendezzik a lista maradékát; végrejátsási ideje  $O(n^2)$ :

```
(* inssort xs = az xs elemeinek a <= reláció szerint
   rendezett listája
   inssort : real list -> real list
   fun inssort (x::xs) = ins(x, inssort xs)
   | inssort [] = []
```

#### 1.1.1. Generikus megoldások

Ha a következő elemet a helyére rakk  $f$  függvény paraméterként adjuk át, az inssort leiszöleges típusú adatok rendezésére használható:

```
(* inssort f xs = az xs elemeinek az f függvény
   segítségével rendezett listája
   inssort : ('a * 'b list -> 'b list) ->
              'a list -> 'b list
*)
fun inssort f (x::xs) = f(x, inssort f xs)
| inssort _ [] = []
```

Még jobb, ha magát az ins függvényt tesszük generikussá:

```
(* ins cmp (x, ys) = az x értékkel a cmp reláció
   szerint
   ins : ('a * 'a -> bool) -> 'a * 'a list -> 'a list
   PRE: ys a cmp reláció szerint rendezett
*)
fun ins cmp (x, ys) =
  let infix cmp
    fun inso (y:ys) =
```

```
    if x cmp y then x::y::ys else y::inso
    ys
    | inso [] = [x]
    inso ys
  end
```

Ezzel inssort egy újabb változata:

```
(* inssort xs = az xs elemeinek a cmp reláció szerint
   rendezett listája
   inssort : ('a * 'a -> bool) -> 'a list -> 'a list
*)
fun inssort cmp (x::xs) = ins cmp (x, inssort cmp xs)
| inssort _ [] = []
```

inssort eddig nem mutatott változatát előbb elemire szét a rendezendő listát, majd hátról visszafele haladva, rendezés közben építik fel az újat. Jobbrekurziót gyűjtőargumentumot használó változatnak kisebb veremre van szüksége, mivel a listáról leválasztott elemeket haladó jobbra haladva azonnal berakja a helyükre az eredménylistában. (A két megoldás futási idéjét az 1.1.3 szakaszban hasonlítjuk össze).

```
fun inssort2 cmp xs =
  (* sort xs zs = az xs már feldolgozott
   elémelnek a cmp reláció szerint
   rendezett listája zs
   sort : 'a list -> 'a list -> 'a list
   *)
  let fun sort (x::xs) zs = sort xs (ins cmp (x,
  xs))
    in sort xs []
  end
```

### 1.1.2. Beszűró rendezés **foldr-relés foldl-rel**

A második argumentumát gyűjtőargumentumként használó foldl sokkal kisebb vermet hozzáj, mint foldr, ezért inssort2 hosszabb listák rendezésére alkalmas. A futási időkről az 1.1.3 szakaszban lesz szó.

```
fun inssort cmp = foldr (ins cmp) []
fun inssort2 cmp = foldl (ins cmp) []
```

### 1.1.3. A futási idők összehasonlítása

Véletlenszerűen el foglalt listák, illetve eredetileg éppen fordított sorrendű listák rendezéséhez szükséges futási idő mérték. Véletlen elosztású listát állít el a Random könyvtábeli rangeList függvény. Növekvő sorrendű egészlistát állít el a -- operátor:

```
infix --;
```

```

fun fm -- to =
  let fun upto to zs = if to < fm
    then zs
    else upto (to-1) (to::zs)
  in upto to []
end;

A futási idők méretéhez 2000 elemet tartalmazó listákat állítunk elő:
val xs = Random.rangeList (1, 100000)
  (2000, Random.newgen());
illetve
val xs = 1 -- 2000;

```

A futási időket az alábbi programmal mérjük meg és fratjuk ki:

```

app Load ["Int", "Random", "Time", "Timer"];
let val starttime = Timer.startCPUTimer();
val zs = insort op>= xs
val {usr=tim,...} = Timer.checkCPUTimer starttime
val t1 = "Integer sort with insort: length = "
  ^ Int.toString(length xs) ^ ", time = "
    ^ Time fmt 2 tim ^ " sec\n";
val starttime = Timer.startCPUTimer();
val zs = insort2 op>= xs
val {usr=tim,...} = Timer.checkCPUTimer starttime
val t2 = "Integer sort with insort2: length = "
  ^ Int.toString(length xs) ^ ", time = "
    ^ Time fmt 2 tim ^ " sec\n"
in
  Bprint(t1 ^ t2)
end;

```

A 2000 elemet tartalmazó, eredetileg fordított sorrendű listák rendezése insort fenti változataival több mint 5-sig, a gyűjtőargumentumos insort 2-ben mutatott változatával viszont csak 0.01-0.02-sig tart (Linux alatt, 233 MHz-es Pentium processzorral). Határolás a nyereség nem beszélhet arról, hogy gyűjtőargumentum nélkül a verem hamar megtelek! De elfinnyük kilönbözések a kétféle változat között, ha ugyanolyan hosszú, de véletlenszerűen elbállított listákat rendezünk: a futási idő bárminyik változat esetén kb. 2 s.

## 1.2. Gyorsrendezés

A gyorsrendezés a növekvő sorrendűben rendezendő sorozatot három részre osztja: az  $m$  mediánra, a mediánnál nem nagyobb, ill. nagyobb elemekre:

$$\boxed{\phantom{m}} \leq \boxed{m} \boxed{\phantom{m}} < \boxed{\phantom{m}}$$

Mediánnak az adott lépéshben rendezendő lista fejét választjuk.

```
(* quicksort cmp xs = az xs elemeinek cmp szerint
rendezett
```

```

quicksort : ('a * 'a -> bool) -> 'a list -> 'a list
*)
fun quicksort cmp xs =
  let (* qs : 'a list -> 'a list
  *)
    fun quicksort cmp xs =
      fun qs (m::ys) = partition ([], [], m, ys)
        | qs ys = ys
      (* partition :
         'a list * 'a list * 'a * 'a
      *)
      list
        -> 'a
      list
        *)
      and partition (ls, rs, m, xs) =
        if cmp(x, m)
        then partition(x::ls, rs, m, xs)
        else partition(ls, x::rs, m, xs)
      | partition (ls, rs, m, []) =
        qs ls @ (m::qs
      rs)
        in
          qs xs
        end
      partition iteratív függvény, amely két eredménylistát épít fel, ls-t és rs-t.
      quicksort a 2000 elemet tartalmazó, eredetileg fordított sorrendű listát majdnem
      14 s alatt rendezi. Az ugyanilyen hosszú, de véletlenszerűen elbállított lista
      rendezéséhez ugyanakkor csak 0.1 s-ra van szüksége.
      A @ műveletet kiküszöbölihetőül járható pl. úgy, hogy az m értékű elemeket egy harmadik
      listában gyűjtjük, amikor a listát két részre bontjuk.

1.3. Összefűsölő rendezés
Az összefűsölő rendezéshez szükségesnek van egy olyan függvényre, amely két listát
növekvő sorrendben egyesít:
(* merge(xs, ys) = xs és ys elemeinek <= szerint
   egyesített listája
merge : int list * int list -> int list
*)
fun merge (xs:xs, ys:ys) =
  if x <= y
  then x::merge(xs, ys:y)
  else y::merge(x::xs, ys)
  | merge ([], ys) = ys
  | merge (xs, []) = xs
vagy ún. réteges minima (...as...) alkalmazásával:
fun merge (xs as x::xs, ys as y::ys) =
  if x <= y
```

```

then x : merge(xs, ys)
else y : merge(xss, ys)
...

```

Hatékonyágról átmenetileg csak a részrendményeket írja ki a veremben tároljuk. Sajnos, ezen nem lehet segíteni, mert iteratív megoldás esetén vissza kellene fordítani az eredményt kapott listát.

### 1.3.1. Fölfelő lefelé haladó összetétele rendezés

A fölfelő lefelé haladó (*top-down*) összetétele rendezés akkor hatékony, ha tözel azonos hosszúságú az a légi lista, amelyekre a rendezendő listát szétszedjük. (A tmsort név elején a t betű utal a *top-down* rendezésre.)

```

(* tmsort xs = az xs elemeinek a <= reláció szerint
   rendezett listája
  *)
fun tmsort xs =
  let val h = Length xs
  in
    if h > 1
    then merge(tmsort(Take(xs, k)),
               tmsort(Drop(xs, k)))
    else xs
  end

```

A legrosszabb esetben  $O(n \cdot \log n)$  lépéstre van szükség. A módszer egyszerű és elég gyors.

### 1.3.2. Alulról fölfelé haladó összetétele rendezés

Az alulról fölfelé haladó (*bottom-up*) összetétele rendezés leggyakrabban változata az eredeti hosszúságú listát  $l$  darab egyenlő hosszúságú listára bontja, majd a szomszédos listákat összelefűti, így 2, 4, 8, 16 stb. elemű listákat állít elő. A megoldás egyszerű, de pazarló. R. O'Keefe algoritmusá (1982) lépései futtatja össze az egyszerű hosszú részlistákat, de csak az utolsó lépésten rendezi az összeset. Az alábbi példában az összelefűtött részlistákat aláhúzással jelöljük:

A	B	C	D	E	F	G	H	I	J	K
<u>A</u>	<u>B</u>	<u>C</u>	<u>D</u>	<u>E</u>	<u>F</u>	<u>G</u>	<u>H</u>	<u>I</u>	<u>J</u>	<u>K</u>
<u>A</u>	<u>B</u>	<u>C</u>	<u>D</u>	<u>E</u>	<u>F</u>	<u>G</u>	<u>H</u>	<u>I</u>	<u>J</u>	<u>K</u>
<u>A</u>	<u>B</u>	<u>C</u>	<u>D</u>	<u>E</u>	<u>F</u>	<u>G</u>	<u>H</u>	<u>I</u>	<u>J</u>	<u>K</u>
<u>A</u>	<u>B</u>	<u>C</u>	<u>D</u>	<u>E</u>	<u>F</u>	<u>G</u>	<u>H</u>	<u>I</u>	<u>J</u>	<u>K</u>
<u>A</u>	<u>B</u>	<u>C</u>	<u>D</u>	<u>E</u>	<u>F</u>	<u>G</u>	<u>H</u>	<u>I</u>	<u>J</u>	<u>K</u>
<u>A</u>	<u>B</u>	<u>C</u>	<u>D</u>	<u>E</u>	<u>F</u>	<u>G</u>	<u>H</u>	<u>I</u>	<u>J</u>	<u>K</u>
<u>A</u>	<u>B</u>	<u>C</u>	<u>D</u>	<u>E</u>	<u>F</u>	<u>G</u>	<u>H</u>	<u>I</u>	<u>J</u>	<u>K</u>

Vagyis az algoritmus először az A-t futtatja össze a B-vel, majd a C-t a D-vel, ezt követően pedig az AB-ot a CD-vel, hiszen most már ezek hossza is egyforma. Ezután E-ot, F-ot, G-ot, H-ot és GH-ot, majd ABCD-öt és EFGH-öt. Ezután E-ot, F-ot, G-ot, H-ot és GH-ot, majd ABCD-öt és EFGH-öt.

```

bmsort néven definiáljuk az összetétele rendezés alulról fölfelé haladó változatát,
amely quicksort-tal kb. azonos idő alatt rendez (a bmsort név elérhető a utal
a bottom-up rendezésre):

(* bmsort xs = az xs elemeinek a <= reláció szerint
   rendezett listája
  *)
fun bmsort xs =
  sorting(xs, [], 0)

bmsort a sorting segédfüggvény használja. Elmek elbőg argumentumuma a
rendezendő lista, második argumentumában a már rendezett részlistákat gyűjtiük
(kezdőértéke []), harmadik argumentum pedig az adott lépéshoz összefuttatandó
elem sorrendjét (kezdetben 0).

Ha a rendezendő lista még nem fogott el, soron következő elemről sorting
egy elemű listát ([x]) kepez, és ez a már rendezett részlisták lista (lss) előtérébe
meghívja a mergepairs segédfüggvényt, megegyezik s, amint látni fogunk, az
argumentumként átadott lista két azonos hosszúságú bal oldali részlistáját fűzi egybe,
feltéve persze, hogy vannek ilyenek. Képben átadtuk elem sorrendzését. Ha a
rendezendő lista különbözik sorting alkészítő lista egyszerűen elemet, a rendezett listát
adják eredményül.

in
  (* sorting(xs, lss, k) = a még rendezetlen xs lista
     elemait berakja a k elemet tartalmazó,
     már rendezett lss listaba *)
  sorting : int list list * int -> int
list
  PRE: k >= 0
  *)
fun sorting (x::xs, lss, k) =
  sorting(xs, mergepairs([x]::lss, k+1), k+1)
  | sorting ([] , lss, k) = hd(mergepairs(lss, 0))

mergepairs egyenlő listában gyűjti a már összelefűtött részlistákat. Nem a lista
hosszát hasonlíta össze, hanem az építen átadott elem k sorrendjából dönti el, hogy
mit kell csinálni a következő részlistával.

(* mergepairs(lss, n) = az n elemet tartalmazó, már
   rendezett lss lista első két
  részlistáját,
  ha egyforma a hosszuk, összelefűtöt
mergepairs : int list list -> int list list
PRE: n >= 0
  *)
fun mergepairs (l1ss as lsl::l2ss::l3ss, n) =
  (* legalább kételemű a lista *)
  if n mod 2 = 1
  then l1ss
  else mergepairs(merge(lsl, l2ss)::l3ss, n div 2)
  | mergepairs (ls, _) = ls (* egyelemű a lista *)

```

Ha n páratlan, mergepairs a listát váltotta a listát vissza, ha pedig páros,
akkor az lss lista elején álló két, egyforma hosszú listát egyetlen rendezett listává



```

smsort : int list -> int list
*)
  fun smsort xs = smsorting(xs, [], 0);

A simarendezséssel szemben a sort néven megaládhába a Listsort könyvtárban.
A függvény specifikációja a következő:
sort ordr xs = xs elemei ordr szerint nem csökkenő
sorrendben (Richard O'Keefe applikatív
simarendezeése)

val sort : ('a * 'a -> order) -> 'a list -> 'a list
Az order típus a General környtári deklarája. compare néven többek között a
Char, az Int, a Real, a String, a Word és a Word8 környtában található olyan
függvény, amely sort elso argumentumaként használható.

A 2000 elemből álló listákat List.sort.sort Int.compare kevesebb mint 0.1
s alatt rendez minden eredetileg fordított sorrendű, mint véletlenszerűen elgállított
listák esetén.

```

**2. Listák használata: polimorf halmazműveletek**

A newMem függvény egy tij elemet rak be egy listába, ha az elem még nincs benne:

```

(* newMem(x, xs) = [x] és xs listaként ábrázolt uniója
newMem : "a * "a list -> "a list
*)
fun newMem (x, xs) = if x isMem xs then xs else x :: xs;
newMem, ha a sorrendőt eltekintünk, halmazhoz létre. A setof függvény halmazt
készít egy listából úgy, hogy kiszedi belőle az ismétlődő elemeket:
(* setof xs = xs elemeinek listaként ábrázolt halmaza
setof : "a list -> "a list
*)
fun setof (x :: xs) = newMem (x, setof xs)
| setof [] = [];

```

A setof függvénynek elég rossz a hatékonysága. Szerencsésekkel, ha a halmazokat a
megszokott halmazműveletekkel kezeljük. Most továbbra is eyszerű listaként
rendezett listát vagy bináris fát.

Öt halmazműveletet definíáltunk: unió (union,  $S \cup T$ ), metsz (inter,  $S \cap T$ ),
részhalmazza-(isSubset,  $T \subseteq S$ ), egyenlők-e (isSetEq,  $S = T$ ), haványhalmaz
(powerset, pS).

```

(* union(xs, ys) = az xs és ys elemeiből álló halmazok
union : "a list * "a list -> "a list
*)

```

(\* inter(xs, ys) = az xs és ys elemeiből álló halmazok
inter : "a list \* "a list -> "a list
\*)

(\* inter([ ], ys) = metszet
inter ([ ], ys) = [];

(\* isInter(xs, ys) = if x isMem ys
then x :: inter(xs, ys)
else inter(xs, ys)
inter ([ ], \_) = [];

(\* isSubset (xs, ys) = az xs elemeiből álló halmaz
részhalmaza-e
isSubset : "a list \* "a list -> bool
\*)

(\* isSubset (x :: xs, ys) = az ys elemeiből álló halmaz
az x isMem ys andalso isSubset(xs, ys)
| isSubset ([ ], \_) = true;
infix isSubset;

A listaek egyenlőségek vizsgálata belső művelet az SML-ben. Halmazokra mégsem
használható, mert pl. a [3, 4] és a [4, 3, 4] listák ugyan különböznek, de mint
halmazok személyében. Halmazként esetleg pl. [3, 4] és [4, 3] is.

```

(* isSetEq(xs, ys) = az xs és ys elemeiből álló
halmazok
isSetEq : "a list * "a list -> bool
*)
fun isSetEq (xs, ys) =
  (xs isSubset ys) andalso (ys isSubset xs);

```

A haványhalmaz egy halmaz összes részhalmazának a halmaza, az eredeti halmazt és
az üres halmazt is beleterve. Leírjuk S-est az eredeti halmazt. S haványhalmazát
úgy állíthatjuk elő, hogy S-ből kiveszünk egy x elemet, és azután rekurzív módon
előállítsuk az  $S - \{x\}$  haványhalmazt. Ha tétszéges  $T$  halmazra  $T' \subseteq S - \{x\}$ ,
akkor  $T \subseteq S$  és  $T \cup \{x\} \subseteq S$ , így minden  $T$ , minden  $T' \cup \{x\}$  eleme  $S'$ 
haványhalmaznak. A pws függvényben a base argumentum gyűjtja a
haványhalmaz elemeit; kezdetben üresnek kell lennie.

```

(* pws(xs, base) = az xs halmaz haványhalmazának
a base halmaznak az uniója
pws : "a list * 'a list -> 'a list list
*)
fun pws (xs, base) =
  fun pws' (xs, base) = pws(xs, base) @ pws(xs,
x :: base)
| pws ([ ], base) = [base];

```

A pws(xs, base) @ pws(xs, x :: base) kitelezésben pws(xs, base)
valósítja neg az  $S - \{x\}$  rekurzív hivást (hiszen  $x :: x$  teljes meg  $S$ -nek), azaz áltja

```

elő az összes olyan halmazt, amelyekben x nincs benne, pws (xs, x : base)
pedig ugyancsak rekurzív módon base-ben gyűjti az x elemeket, vagyis elbővíti az
összes olyan halmazt, amelyben x benne van. Halmazegyenlettel pws eredménye így
adható meg:
pws(S, B) = {T ∪ BT | T ⊆ S}
(* powerset xs = az xs halmaz hatványhalmaza
powerset : 'a list -> 'a list list
*)
fun powerset xs = pws(xs, []);

```

Az algoritmus helyességét nem könnyű belátni. A tén. *magasabb rendű* függvények tárgyalásakor majd könyebbben igazolható megoldást mutatunk be.

### 3. További rekurzív függvények

Egy függvény  $n$ -edik hatványát így specifikálhatjuk:  $f^n = f(\dots(f(x))\dots)$ , ha  $n \geq 0$  ( $f$ -n-szer ismétlőik). Definiálunk SML-függvényt ilyen ismétlések feltrására!

```

(* repeat f n x = f n-edik hatványa az x helyen
repeat : ('a -> 'a) -> int -> 'a -> 'a
*)
fun repeat f n x =
  if n > 0 then repeat f (n-1) (f x) else x

```

Sok függvény feliratához repeat segítségével. Példák:

```

(* drop(xs, k) = xs első k elemének elhagyássával
drop : 'a list * int -> 'a list
*)
fun drop (xs, k) = repeat tl k xs
(* replist k = füzér, amelyben "Ha!" k-szor ismétlődik
replist : int -> string list
*)
fun replist k = repeat (secl "Ha!" op::) k []

```

### 4. A case-kifejezés

Szintaxisa a következő:

```

case E of P1 => E1 | P2 => E2 | ... | Pn => En
Az SML-értelmező – balról jobbra és ittől lefelé haladva – megszabálya E-t P1-re
illeszteni, ha nem sikerül, P2-re s.t. A case-kifejezés eredménye az E kifejezésre
illesztései első Pi minthoz tartozó Ei kifejezés lesz. Például a lady függvényt így
is definíálhatunk volna:
(* lady p = p fönemes hitvesénék rangja
lady : degree -> string
*)

```