FUTÁSI IDŐ MÉRÉSE

A futási idők összehasonlítása

- 2000 elemet tartalmazó, véletlenszerűen előállított, illetve eredetileg éppen fordított sorrendű listák rendezéséhez szükséges futási időt mérünk.
  - Véletlen eloszlású egészlistát állít elő a `Random` könyvtárbeli `rangelist` függvény:

```
val xs2000R =
      Random.rangelist (1, 100000) (2000, Random.newgen());
```

  - Növekvő sorrendű egészlistát állít elő 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;

val xs2000N = 1 -- 2000;
```

A futási idők összehasonlítása (folyt.)

- A futási időt az alábbi függvénnyel mérhetjük:

```
fun futIdo (sort, sortFn) (cmp, cmpFn) (xs, kind) =
      let val starttime = Timer.startCPUTimer()
          val zs = sort cmp xs
          val usr=tim,... = Timer.checkCPUTimer starttime
      in
          "Int sort with " ^ sortFn ^ ", " ^ cmpFn ^
          ", length = " ^ Int.toString(length xs) ^ " (" ^
          kind ^ "), time = " ^ Time.fmt 2 tim ^ " sec\n"
      end;
```

```
val t1N = futIdo (inssort, "inssort") (op>=, "op>=") (xs2000N, "increasing");
val t2N = futIdo (inssort2, "inssort2") (op>=, "op>=") (xs2000N, "increasing");
val t1R = futIdo (inssort, "inssort") (op>=, "op>=") (xs2000R, "random");
val t2R = futIdo (inssort2, "inssort2") (op>=, "op>=") (xs2000R, "random");
```


A futási idők összehasonlítása (folyt.)

- A 2000 elemű, fordított sorrendű lista rendezése az akkumulátort nem használó `inssort`-változatokkal több mint 5 s-ig, az akkumulátort használó változatokkal csak 0.01 s-ig tart (linux, 233 MHz-es Pentium).

```
Int sort with inssort, op>=, length = 2000 (increasing), time = 5.18 sec
Int sort with inssort2, op>=, length = 2000 (increasing), time = 0.01 sec
Int sort with inssortRi, op>=, length = 2000 (increasing), time = 5.14 sec
Int sort with inssortLi, op>=, length = 2000 (increasing), time = 0.01 sec
```

- Eltűnik a különbség, ha ugyanolyan hosszú, de véletlenszerűen előállított listákat rendezünk.

```
Int sort with inssort, op>=, length = 2000 (random), time = 2.39 sec
Int sort with inssort2, op>=, length = 2000 (random), time = 2.26 sec
Int sort with inssortRi, op>=, length = 2000 (random), time = 2.40 sec
Int sort with inssortLi, op>=, length = 2000 (random), time = 2.24 sec
```

LISTÁK RENDEZÉSE

Listák rendezése

- `inssort` (beszúró rendezés),
- **`selsort` (kiválasztó rendezés),**
- **`quicksort` (gyorsrendezés),**
- **`tmsort` (felülről lefelé haladó összefésülő rendezés),**
- `bmsort` (alulról felfelé haladó összefésülő rendezés),
- `smsort` (simarendezés).

Az `order` típus

Az `order` típus definíciója (ld. `General.sig`

```
datatype order = LESS | EQUAL | GREATER

[order] is used as the return type of comparison functions.
```

Példák az SML-alapkönyvtárból (SML Basis Library)

```
Int.compare     : int * int -> order
Char.compare    : char * char -> order
Real.compare    : real * real -> order
String.compare  : string * string -> order
Time.compare    : time * time -> order
```


Kiválasztó rendezés

```
(* selsort : ('a * 'a -> order) -> 'a list -> 'a list
   selsort cmp xs = az xs elemei cmp szerint növekvő sorrendben
*)
fun selsort cmp xs =
      let

          (* max : 'a * 'a -> 'a
             max (x, y) = x és y közül cmp szerint a nagyobb
          *)
          fun max (x, y) = if cmp(x, y) = GREATER then x else y

          (* min : 'a * 'a -> 'a
             min (x, y) = x és y közül cmp szerint a kisebb
          *)
          fun min (x, y) = if cmp(x, y) = LESS then x else y
```

Kiválasztó rendezés (folyt.)

```
(* maxSelect : 'a * 'a list * 'a list -> 'a * 'a list
   maxSelect (x, ys, zs) = pár, amelynek első tagja az
      (x::ys) cmp szerinti legnagyobb eleme, második
      tagja az x::ys többi eleméből és a zs
      elemeiből álló lista                              *)
fun maxSelect (x, [], zs) = (x, zs)
  | maxSelect (x, y::ys, zs) =
                maxSelect(max(x, y), ys, min(x,y)::zs)

(* sSort : 'a list * 'a list -> 'a list
   sSort (xs, ws) = az xs elemei cmp szerint növekvő
                sorrendben a ws elé fűzve               *)
fun sSort ([], ws) = ws
  | sSort (x::xs, ws) =
      let val (z, zs) = maxSelect(x, xs, [])
      in
         sSort (zs, z::ws)
      end
```


Kiválasztó rendezés (folyt.)

```
    in
        sSort (xs, [])
    end

app load ["Int","Char","Real"];

selsort Int.compare [1,2,3,4,5,6,7,8,9];
selsort Int.compare [9,8,7,6,5,4,3,2,1];
selsort Real.compare [4.5,6.7,3.6,4.3,1.2,0.9,8.9,9.8,2.0];
selsort Char.compare (explode "Ej mi a ko tyukanyo");
```

Gyorsrendezés, akkumulátor használata nélkül

```
(* quicksort1 cmp xs = az xs elemeinek cmp szerint rendezett listája
   quicksort1 : ('a * 'a -> order) -> 'a list -> 'a list
*)
fun quicksort1 cmp xs =
      let (* qs : 'a list -> 'a list
             qs ys = az ys elemeinek cmp szerint rendezett listája *)
          fun qs (m::ys) =
                let (* partition : 'a list *  'a list * ' list -> 'a list
                       partition (xs, ls, rs) = ... *)
                    fun partition (x::xs, ls, rs) =
                          if cmp(x, m) = LESS then partition(xs, x::ls, rs)
                                             else partition(xs, ls, x::rs)
                      | partition ([], ls, rs) = qs ls @ (m::qs rs)
                in
                    partition (ys, [], [])
                end
            | qs [] = []
      in
         qs xs
      end;
```


Gyorsrendezés, akkumulátor használatával

```
(* quicksort2 cmp xs = az xs elemeinek cmp szerint rendezett listája
   quicksort2 : ('a * 'a -> order) -> 'a list -> 'a list
*)
fun quicksort2 cmp xs =
      let (* qs : 'a list -> 'a list
             qs ys = az ys elemeinek cmp szerint rendezett listája  *)
          fun qs (m::ys) zs =
                let (* partition : 'a list * a' list * 'a list -> 'a list
                       partition (xs, ls, rs) = ... *)
                    fun partition (x::xs, ls, rs) =
                          if cmp(x, m) = LESS then partition(xs, x::ls, rs)
                                             else partition(xs, ls, x::rs)
                      | partition ([], ls, rs) = qs ls (m :: qs rs zs)
                in
                    partition (ys, [], [])
                end
            | qs [] zs = zs
      in
          qs xs []
      end;
```

A futási idők összehasonlítása

```
val t1 = futIdo (inssort2, "inssort2") (op>=, "op>=") (xs2000R, "random");
                                                  (* ~ 2 M összehasonlítás! *)
val t3 = futIdo (quicksort2, "quicksort2")
                (Int.compare, "Int.compare") (xs20000R, "random");
val t4 = futIdo (Listsort.sort, "Listsort.sort")
                (Int.compare, "Int.compare") (xs20000R, "random");
                                                  (* ~ 300 E összehasonlítás *)

Int sort with inssort2, op>=, length = 2000 (random), time = 2.30 sec

Int sort with quicksort1, Int.compare, length = 20000 (random), time = 2.18 sec
Int sort with quicksort2, Int.compare, length = 20000 (random), time = 1.72 sec
Int sort with Listsort.sort, Int.compare, length = 20000 (random), time = 1.76 sec

Int sort with quicksort2, Int.compare, length = 200000 (random), time = 27.13 sec
Int sort with quicksort1, Int.compare, length = 200000 (random), time = 32.59 sec

val t7 = futIdo (Listsort.sort, "Listsort.sort") (Int.compare, "Int.compare")
         (Random.rangelist (1, 100000) (200000, Random.newgen()), "random");
! Uncaught exception:
! Out_of_memory
```


Összefésülő rendezések

- Az összefésülő rendezéshez kell egy olyan függvény, 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 (xxs as x::xs, yys as y::ys)=
      if x <= y
      then x::merge(xs, yys)
      else y::merge(xxs, ys)
  | merge ([], ys) = ys
  | merge (xs, []) = xs;
```

- Hatékonyságromlást okoz, hogy a részeredményeket a veremben tároljuk. Iteratív megoldás esetén meg kell fordítani az eredménylistát.

Fölülről lefelé haladó összefésülő rendezés

- A fölülről lefelé haladó összefésülő rendezés *(top-down merge sort)* akkor hatékony, ha közel azonos hosszúságú az a két lista, amelyekre a rendezendő listát szétszedjük.

```
(* tmsort xs = az xs elemeinek a <= reláció szerint
               rendezett listája
   tmsort : int list -> int list
*)
fun tmsort xs = let val h = length xs
                    val k = h div 2
                in
                   if h > 1
                   then merge(tmsort(List.take(xs, k)),
                              tmsort(List.drop(xs, k)))
                   else xs
                end;
```

- A legrosszabb esetben $O(n \cdot log\, n)$ lépésre van szükség.