```
structure ArraySequence : SEQUENCE =
struct
  (* An array with starting index and length *)
  type 'a seq = { ary : 'a array, idx : int, len : int }
  type 'a ord = 'a * 'a -> order

  datatype 'a treeview = EMPTY | ELT of 'a | NODE of 'a seq * 'a ↩
      seq
  datatype 'a listview = NIL | CONS of 'a * 'a seq

  exception Range
  exception Size

  val length : 'a seq -> int = #len
  fun empty _ = {ary=Array.fromList [], idx=0, len=0}
  fun singleton x = {ary=Array.fromList [x], idx=0, len=1}

  fun collate cmp ({ary=x, ...} : 'a seq, {ary=y, ...} : 'a seq) ↩
      =
      Array.collate cmp (x, y)

  fun tabulate f n =
      if n < 0 then raise Size
      else {ary=Array.tabulate (n, f), idx=0, len=n}

  fun nth {ary, idx, len} i =
      if i < 0 orelse i >= len then raise Range
      else Array.sub (ary, idx+i)

  fun toString f s =
      "<" ^ String.concatWith "," (List.tabulate (length s, f o ↩
          nth s)) ^ ">"

  fun fromList l =
      let val ary = Array.fromList l
      in {ary=ary, idx=0, len=Array.length ary}
      end

  val % = fromList

  fun subseq {ary, idx, len} (i, len') =
      if len' < 0 then raise Size
      else if i < 0 orelse i+len' > len then raise Range
      else {ary=ary, idx=idx+i, len=len'}

  fun take (s, n) = subseq s (0, n)
  fun drop (s, n) = subseq s (n, length s - n)

  fun showl {len=0, ...} = NIL
    | showl s = CONS (nth s 0, drop (s, 1))

  fun showt s =
      case length s
        of 0 => EMPTY
         | 1 => ELT (nth s 0)
         | n => let val half = n div 2
                in NODE (take (s, half), drop (s, half))
```

```
              end

fun rev s =
    let val n = length s
    in tabulate (fn i => nth s (n-1-i)) n
    end

fun append (s, t) =
    let val (ns, nt) = (length s, length t)
      fun ith i = if i >= ns then nth t (i-ns) else nth s i
    in tabulate ith (ns+nt)
    end

fun iterh f b s =
    let
      fun iterh' s (old, cur) =
          case showl s
            of NIL => (rev (fromList old), cur)
             | CONS (x, xs) => iterh' xs (cur::old, f (cur, x))
    in iterh' s ([], b)
    end

fun iter f b s = #2 (iterh f b s)
fun toList s = iter (fn (l,x) => x::l) [] (rev s)

fun merge cmp s t =
    let
      (* Sequential merge. Pretend it's parallel! *)
      fun merge' s t =
          case (showl s, showl t)
            of (NIL, _) => toList t
             | (_, NIL) => toList s
             | (CONS (x, xs), CONS (y, ys)) =>
               if cmp (y, x) = LESS then y::merge' s ys
               else x::merge' xs t

    in fromList (merge' s t)
    end

fun sort cmp s =
    case showt s
      of EMPTY => s
       | ELT x => singleton x
       | NODE (l, r) => merge cmp (sort cmp l) (sort cmp r)

fun enum s = tabulate (fn i => (i, nth s i)) (length s)
fun map f s = tabulate (f o (nth s)) (length s)
fun map2 f s t =
    tabulate (fn i => f (nth s i, nth t i)) (Int.min (length s,↩
        length t))
fun unmap2 (spl : 'a -> 'b * 'c) s =
    let
      val n = length s
      val s' = map spl s
    in (tabulate (#1 o nth s') n, tabulate (#2 o nth s') n)
    end
fun zip s t = map2 (fn x => x) s t
```

```
fun unzip s = unmap2 (fn x => x) s

fun reduce f b s =
    if length s = 0 then b
    else let
      fun pp2 x n = if n >= x then n div 2 else pp2 x (2*n)
      fun prevPow2 x = pp2 x 1
      fun reduce' s =
          case length s
            of 1 => nth s 0
             | n => let val half = prevPow2 n
                    in f (reduce' (take (s, half)),
                          reduce' (drop (s, half)))
                    end
    in f (b, reduce' s)
    end

fun scan f b s =
    case length s
     of 0 => (empty (), b)
      | 1 => (singleton b, f (b, nth s 0))
      | n =>
        let
          fun contract i =
              if i = n div 2 then nth s (2*i)
              else f (nth s (2*i), nth s (2*i+1))
          val s' = tabulate contract (((n-1) div 2) + 1)
          val (r, res) = scan f b s'
          fun expand i =
              if i mod 2 = 0 then nth r (i div 2)
              else f (nth r (i div 2), nth s (i-1))
        in (tabulate expand n, res)
        end

fun scani f b s =
    let val (r, res) = scan f b s
    in append (r, singleton res)
    end

infix >>
fun ({ary, ...} : 'a seq) >> (f : int * 'a -> unit) =
    Array.appi f ary

fun flatten ss =
    let
      val (starts, n) = scan op+ 0 (map length ss)
      val idxs = tabulate (fn _ => (0, 0)) n
      fun setIdxs (i, (start,s)) =
          s >> (fn (j,_) => Array.update (#ary idxs, start+j, (↩
              i,j)))
      val _ = zip starts ss >> setIdxs
      fun ith i =
          let val (j, k) = nth idxs i
          in nth (nth ss j) k
          end
    in tabulate ith n
    end
```

```
fun filter p s =
    flatten (map (fn x => if p x then singleton x else empty ()↩
        ) s)

fun argmax cmp s =
    case length s
      of 0 => raise Range
       | n =>
         let
           fun best (i, j) =
               case cmp (nth s j, nth s i)
                 of LESS => j
                  | _ => i
         in reduce best 0 (tabulate (fn i => i) n)
         end

fun inject idx s =
    let val s' as {ary, ...} = tabulate (nth s) (length s)
      val _ = idx >> (fn (_,(i,x)) => Array.update (ary, i, x))
    in s'
    end

local
  datatype 'a diff = RUN of int | SAME of int

  fun opdiff (_, RUN x) = RUN x
    | opdiff (RUN x, SAME y) = RUN (x+y)
    | opdiff (SAME x, SAME y) = SAME (x+y)
in
  fun collect cmp s =
      case length s
        of 0 => empty ()
         | n =>
           let
             val (ks, vs) = unzip (sort (fn ((x,_), (y,_)) => ↩
                 cmp (x,y)) s)

             (* SAME 1 indicates the next position has the same↩
                  key *)
             fun dk (x,y) = if cmp (x,y) = EQUAL then SAME 1 ↩
                 else RUN 1
             val diffs = map2 dk (take (ks, n-1)) (drop (ks, 1)↩
                 )
             val runs = scani opdiff (RUN 1) diffs

             (* RUN 1 marks the starting position of each ↩
                 sequence *)
             val flags = map (fn (i, RUN 1) => SOME i | _ => ↩
                 NONE) (enum runs)
             val starts = map valOf (filter isSome flags)
             val lengths = map2 op- (drop (append (starts, %[n↩
                 ]), 1)) starts

             fun make (i, len) =
                 let val k = nth ks i
                 in (k, subseq vs (i, len))
```

```
                    end
                in map2 make starts lengths
                end
  end
end
```