Chapter B8. Sorting

Caution! If you are expecting to sort efficiently on a parallel machine, whether its parallelism is small-scale or massive, you almost certainly want to use library routines that are specific to your hardware.

We include in this chapter translations into Fortran 90 of the general purpose *serial* sorting routines that are in Volume 1, augmented by several new routines that give pedagogical demonstrations of how parallel sorts can be achieved with Fortran 90 parallel constructions and intrinsics. However, we intend the above word "pedagogical" to be taken seriously: these new, supposedly parallel, routines are *not* likely to be competitive with machine-specific library routines. Neither do they compete successfully on serial machines with the all-serial routines provided (namely `sort`, `sort2`, `sort3`, `indexx`, and `select`).

⋆ ⋆ ⋆

```
SUBROUTINE sort_pick(arr)
USE nrtype
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: arr
   Sorts an array arr into ascending numerical order, by straight insertion. arr is replaced
   on output by its sorted rearrangement.
INTEGER(I4B) :: i,j,n
REAL(SP) :: a
n=size(arr)
do j=2,n                        Pick out each element in turn.
    a=arr(j)
    do i=j-1,1,-1               Look for the place to insert it.
        if (arr(i) <= a) exit
        arr(i+1)=arr(i)
    end do
    arr(i+1)=a                  Insert it.
end do
END SUBROUTINE sort_pick
```

Not only is `sort_pick` (renamed from Volume 1's `piksrt`) *not parallelizable*, but also, even worse, it is an N^2 routine. It is meant to be invoked only for the most trivial sorting jobs, say, $N < 20$.

⋆ ⋆ ⋆

```
SUBROUTINE sort_shell(arr)
USE nrtype
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: arr
    Sorts an array arr into ascending numerical order by Shell's method (diminishing increment
    sort). arr is replaced on output by its sorted rearrangement.
INTEGER(I4B) :: i,j,inc,n
REAL(SP) :: v
n=size(arr)
inc=1
do                              Determine the starting increment.
    inc=3*inc+1
    if (inc > n) exit
end do
do                              Loop over the partial sorts.
    inc=inc/3
    do i=inc+1,n                Outer loop of straight insertion.
        v=arr(i)
        j=i
        do                      Inner loop of straight insertion.
            if (arr(j-inc) <= v) exit
            arr(j)=arr(j-inc)
            j=j-inc
            if (j <= inc) exit
        end do
        arr(j)=v
    end do
    if (inc <= 1) exit
end do
END SUBROUTINE sort_shell
```

The routine sort_shell is renamed from Volume 1's shell. Shell's Method, a diminishing increment sort, is not directly parallelizable. However, one can write a fully parallel routine (though not an especially fast one — see remarks at beginning of this chapter) in much the same spirit:

```
SUBROUTINE sort_byreshape(arr)
USE nrtype; USE nrutil, ONLY : swap
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: arr
    Sort an array arr by bubble sorting a succession of reshapings into array slices. The method
    is similar to Shell sort, but allows parallelization within the vectorized masked swap calls.
REAL(SP), DIMENSION(:,:), ALLOCATABLE :: tab
REAL(SP), PARAMETER :: big=huge(arr)
INTEGER(I4B) :: inc,n,m
n=size(arr)
inc=1
do                                      Find the largest increment that fits.
    inc=2*inc+1
    if (inc > n) exit
end do
do                                      Loop over the different shapes for the reshaped
    inc=inc/2                               array.
    m=(n+inc-1)/inc
    allocate(tab(inc,m))                Allocate space and reshape the array. big en-
    tab=reshape(arr, (/inc,m/) , (/big/) )      sures that fill elements stay at the
    do                                          end.
          Bubble sort all the rows in parallel.
        call swap(tab(:,1:m-1:2),tab(:,2:m:2), &
            tab(:,1:m-1:2)>tab(:,2:m:2))
        call swap(tab(:,2:m-1:2),tab(:,3:m:2), &
            tab(:,2:m-1:2)>tab(:,3:m:2))
```

```
        if (all(tab(:,1:m-1) <= tab(:,2:m))) exit
    end do
    arr=reshape(tab,shape(arr))        Put the array back together for the next shape.
    deallocate(tab)
    if (inc <= 1) exit
end do
END SUBROUTINE sort_byreshape
```

The basic idea is to reshape the given one-dimensional array into a succession of two-dimensional arrays, starting with "tall and narrow" (many rows, few columns), and ending up with "short and wide" (many columns, few rows). At each stage we sort all the rows in parallel by a bubble sort, giving something close to Shell's diminishing increments.

⋆ ⋆ ⋆

We now arrive at those routines, based on the Quicksort algorithm, that we actually intend for use with general N on serial machines:

```
SUBROUTINE sort(arr)
USE nrtype; USE nrutil, ONLY : swap,nrerror
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: arr
INTEGER(I4B), PARAMETER :: NN=15, NSTACK=50
    Sorts an array arr into ascending numerical order using the Quicksort algorithm. arr is
    replaced on output by its sorted rearrangement.
    Parameters: NN is the size of subarrays sorted by straight insertion and NSTACK is the
    required auxiliary storage.
REAL(SP) :: a
INTEGER(I4B) :: n,k,i,j,jstack,l,r
INTEGER(I4B), DIMENSION(NSTACK) :: istack
n=size(arr)
jstack=0
l=1
r=n
do
    if (r-l < NN) then                 Insertion sort when subarray small enough.
        do j=l+1,r
            a=arr(j)
            do i=j-1,l,-1
                if (arr(i) <= a) exit
                arr(i+1)=arr(i)
            end do
            arr(i+1)=a
        end do
        if (jstack == 0) RETURN
        r=istack(jstack)               Pop stack and begin a new round of partition-
        l=istack(jstack-1)                 ing.
        jstack=jstack-2
    else                               Choose median of left, center, and right elements
        k=(l+r)/2                          as partitioning element a. Also rearrange so
        call swap(arr(k),arr(l+1))         that a(l) ≤ a(l+1) ≤ a(r).
        call swap(arr(l),arr(r),arr(l)>arr(r))
        call swap(arr(l+1),arr(r),arr(l+1)>arr(r))
        call swap(arr(l),arr(l+1),arr(l)>arr(l+1))
        i=l+1                          Initialize pointers for partitioning.
        j=r
        a=arr(l+1)                     Partitioning element.
        do                             Here is the meat.
            do                         Scan up to find element >= a.
                i=i+1
```

```
            if (arr(i) >= a) exit
        end do
        do                              Scan down to find element <= a.
            j=j-1
            if (arr(j) <= a) exit
        end do
        if (j < i) exit                 Pointers crossed. Exit with partitioning complete.
        call swap(arr(i),arr(j))            Exchange elements.
      end do
      arr(l+1)=arr(j)                   Insert partitioning element.
      arr(j)=a
      jstack=jstack+2
        Push pointers to larger subarray on stack; process smaller subarray immediately.
      if (jstack > NSTACK) call nrerror('sort: NSTACK too small')
      if (r-i+1 >= j-l) then
          istack(jstack)=r
          istack(jstack-1)=i
          r=j-1
      else
          istack(jstack)=j-1
          istack(jstack-1)=l
          l=i
      end if
    end if
end do
END SUBROUTINE sort
```

f90 `call swap(...) ... call swap(...)` One might think twice about putting all these external function calls (to `nrutil` routines) in the inner loop of something as streamlined as a sort routine, but here they are executed only once for each partitioning.

`call swap(arr(i),arr(j))` This call *is* in a loop, but not the innermost loop. Most modern machines are very fast at the "context changes" implied by subroutine calls and returns; but in a time-critical context you might code this swap in-line and see if there is any timing difference.

```
SUBROUTINE sort2(arr,slave)
USE nrtype; USE nrutil, ONLY : assert_eq
USE nr, ONLY : indexx
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: arr,slave
    Sorts an array arr into ascending order using Quicksort, while making the corresponding
    rearrangement of the same-size array slave. The sorting and rearrangement are performed
    by means of an index array.
INTEGER(I4B) :: ndum
INTEGER(I4B), DIMENSION(size(arr)) :: index
ndum=assert_eq(size(arr),size(slave),'sort2')
call indexx(arr,index)              Make the index array.
arr=arr(index)                      Sort arr.
slave=slave(index)                  Rearrange slave.
END SUBROUTINE sort2
```

⋆ ⋆ ⋆

A close surrogate for the Quicksort partition-exchange algorithm can be coded, parallelizable, by using Fortran 90's `pack` intrinsic. On real compilers, unfortunately, the resulting code is not very efficient as compared with (on serial machines) the tightness of `sort`'s inner loop, above, or (on parallel machines) supplied library sort routines. We illustrate the principle nevertheless in the following routine.

```
RECURSIVE SUBROUTINE sort_bypack(arr)
USE nrtype; USE nrutil, ONLY : array_copy,swap
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: arr
```
Sort an array arr by recursively applying the Fortran 90 pack intrinsic. The method is similar to Quicksort, but this variant allows parallelization by the Fortran 90 compiler.
```
REAL(SP) :: a
INTEGER(I4B) :: n,k,nl,nerr
INTEGER(I4B), SAVE :: level=0
LOGICAL, DIMENSION(:), ALLOCATABLE, SAVE :: mask
REAL(SP), DIMENSION(:), ALLOCATABLE, SAVE :: temp
n=size(arr)
if (n <= 1) RETURN
k=(1+n)/2
call swap(arr(1),arr(k),arr(1)>arr(k))          Pivot element is median of first, middle,
call swap(arr(k),arr(n),arr(k)>arr(n))              and last.
call swap(arr(1),arr(k),arr(1)>arr(k))
if (n <= 3) RETURN
level=level+1                                   Keep track of recursion level to avoid al-
if (level == 1) allocate(mask(n),temp(n))           location overhead.
a=arr(k)
mask(1:n) = (arr <= a)                          Which elements move to left?
mask(k) = .false.
call array_copy(pack(arr,mask(1:n)),temp,nl,nerr)       Move them.
mask(k) = .true.
temp(nl+2:n)=pack(arr,.not. mask(1:n))          Move others to right.
temp(nl+1)=a
arr=temp(1:n)
call sort_bypack(arr(1:nl))                     And recurse.
call sort_bypack(arr(nl+2:n))
if (level == 1) deallocate(mask,temp)
level=level-1
END SUBROUTINE sort_bypack
```

⋆ ⋆ ⋆

The following routine, sort_heap, is renamed from Volume 1's hpsort.

```
SUBROUTINE sort_heap(arr)
USE nrtype
USE nrutil, ONLY : swap
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: arr
```
Sorts an array arr into ascending numerical order using the Heapsort algorithm. arr is replaced on output by its sorted rearrangement.
```
INTEGER(I4B) :: i,n
n=size(arr)
do i=n/2,1,-1
```
The index i, which here determines the "left" range of the sift-down, i.e., the element to be sifted down, is decremented from n/2 down to 1 during the "hiring" (heap creation) phase.
```
    call sift_down(i,n)
end do
do i=n,2,-1
```
Here the "right" range of the sift-down is decremented from n−1 down to 1 during the "retirement-and-promotion" (heap selection) phase.
```
    call swap(arr(1),arr(i))                    Clear a space at the end of the array, and
    call sift_down(1,i-1)                           retire the top of the heap into it.
end do
CONTAINS

SUBROUTINE sift_down(l,r)
INTEGER(I4B), INTENT(IN) :: l,r
```

```
    Carry out the sift-down on element arr(l) to maintain the heap structure.
INTEGER(I4B) :: j,jold
REAL(SP) :: a
a=arr(l)
jold=l
j=l+l
do                                          "Do while j <= r:"
    if (j > r) exit
    if (j < r) then
        if (arr(j) < arr(j+1)) j=j+1        Compare to the better underling.
    end if
    if (a >= arr(j)) exit                   Found a's level. Terminate the sift-down. Oth-
    arr(jold)=arr(j)                            erwise, demote a and continue.
    jold=j
    j=j+j
end do
arr(jold)=a                                 Put a into its slot.
END SUBROUTINE sift_down
END SUBROUTINE sort_heap
```

⋆ ⋆ ⋆

Another opportunity provided by Fortran 90 for a fully parallelizable sort, at least pedagogically, is to use the language's allowed access to the actual floating-point representation and to code a radix sort [1] on its bits. This is *not* efficient, but it illustrates some Fortran 90 language features perhaps worthy of study for other applications.

```
SUBROUTINE sort_radix(arr)
USE nrtype; USE nrutil, ONLY : array_copy,nrerror
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: arr
    Sort an array arr by radix sort on its bits.
INTEGER(I4B), DIMENSION(size(arr)) :: narr,temp
LOGICAL, DIMENSION(size(arr)) :: msk
INTEGER(I4B) :: k,negm,ib,ia,n,nl,nerr
  Because we are going to transfer reals to integers, we must check that the number of bits
  is the same in each:
ib=bit_size(narr)
ia=ceiling(log(real(maxexponent(arr)-minexponent(arr),sp))/log(2.0_sp)) &
    + digits(arr)
if (ib /= ia) call nrerror('sort_radix: bit sizes not compatible')
negm=not(ishftc(1,-1))                              Mask for all bits except sign bit.
n=size(arr)
narr=transfer(arr,narr,n)
where (btest(narr,ib-1)) narr=ieor(narr,negm)       Flip all bits on neg. numbers.
do k=0,ib-2
      Work from low- to high-order bits, and partition the array according to the value of the
      bit.
    msk=btest(narr,k)
    call array_copy(pack(narr,.not. msk),temp,nl,nerr)
    temp(nl+1:n)=pack(narr,msk)
    narr=temp
end do
msk=btest(narr,ib-1)                                The sign bit gets separate treat-
call array_copy(pack(narr,msk),temp,nl,nerr)            ment, since here 1 comes be-
temp(nl+1:n)=pack(narr,.not. msk)                       fore 0.
narr=temp
where (btest(narr,ib-1)) narr=ieor(narr,negm)       Unflip all bits on neg. numbers.
arr=transfer(narr,arr,n)
END SUBROUTINE sort_radix
```

⋆ ⋆ ⋆

f90 We overload the generic name indexx with two specific implementations, one for SP floating values, the other for I4B integers. (You can of course add more overloadings if you need them.)

```
SUBROUTINE indexx_sp(arr,index)
USE nrtype; USE nrutil, ONLY : arth,assert_eq,nrerror,swap
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: arr
INTEGER(I4B), DIMENSION(:), INTENT(OUT) :: index
INTEGER(I4B), PARAMETER :: NN=15, NSTACK=50
    Indexes an array arr, i.e., outputs the array index of length N such that arr(index(j))
    is in ascending order for j = 1, 2, ..., N. The input quantity arr is not changed.
REAL(SP) :: a
INTEGER(I4B) :: n,k,i,j,indext,jstack,l,r
INTEGER(I4B), DIMENSION(NSTACK) :: istack
n=assert_eq(size(index),size(arr),'indexx_sp')
index=arth(1,1,n)
jstack=0
l=1
r=n
do
    if (r-l < NN) then
        do j=l+1,r
            indext=index(j)
            a=arr(indext)
            do i=j-1,l,-1
                if (arr(index(i)) <= a) exit
                index(i+1)=index(i)
            end do
            index(i+1)=indext
        end do
        if (jstack == 0) RETURN
        r=istack(jstack)
        l=istack(jstack-1)
        jstack=jstack-2
    else
        k=(l+r)/2
        call swap(index(k),index(l+1))
        call icomp_xchg(index(l),index(r))
        call icomp_xchg(index(l+1),index(r))
        call icomp_xchg(index(l),index(l+1))
        i=l+1
        j=r
        indext=index(l+1)
        a=arr(indext)
        do
            do
                i=i+1
                if (arr(index(i)) >= a) exit
            end do
            do
                j=j-1
                if (arr(index(j)) <= a) exit
            end do
            if (j < i) exit
            call swap(index(i),index(j))
        end do
        index(l+1)=index(j)
        index(j)=indext
        jstack=jstack+2
        if (jstack > NSTACK) call nrerror('indexx: NSTACK too small')
        if (r-i+1 >= j-l) then
            istack(jstack)=r
```

```
            istack(jstack-1)=i
            r=j-1
        else
            istack(jstack)=j-1
            istack(jstack-1)=l
            l=i
        end if
    end if
end do
CONTAINS
SUBROUTINE icomp_xchg(i,j)
INTEGER(I4B), INTENT(INOUT) :: i,j
INTEGER(I4B) :: swp
if (arr(j) < arr(i)) then
    swp=i
    i=j
    j=swp
end if
END SUBROUTINE icomp_xchg
END SUBROUTINE indexx_sp



SUBROUTINE indexx_i4b(iarr,index)
USE nrtype; USE nrutil, ONLY : arth,assert_eq,nrerror,swap
IMPLICIT NONE
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: iarr
INTEGER(I4B), DIMENSION(:), INTENT(OUT) :: index
INTEGER(I4B), PARAMETER :: NN=15, NSTACK=50
INTEGER(I4B) :: a
INTEGER(I4B) :: n,k,i,j,indext,jstack,l,r
INTEGER(I4B), DIMENSION(NSTACK) :: istack
n=assert_eq(size(index),size(iarr),'indexx_sp')
index=arth(1,1,n)
jstack=0
l=1
r=n
do
    if (r-l < NN) then
        do j=l+1,r
            indext=index(j)
            a=iarr(indext)
            do i=j-1,l,-1
                if (iarr(index(i)) <= a) exit
                index(i+1)=index(i)
            end do
            index(i+1)=indext
        end do
        if (jstack == 0) RETURN
        r=istack(jstack)
        l=istack(jstack-1)
        jstack=jstack-2
    else
        k=(l+r)/2
        call swap(index(k),index(l+1))
        call icomp_xchg(index(l),index(r))
        call icomp_xchg(index(l+1),index(r))
        call icomp_xchg(index(l),index(l+1))
        i=l+1
        j=r
        indext=index(l+1)
        a=iarr(indext)
        do
            do
```

```
            i=i+1
            if (iarr(index(i)) >= a) exit
        end do
        do
            j=j-1
            if (iarr(index(j)) <= a) exit
        end do
        if (j < i) exit
        call swap(index(i),index(j))
    end do
    index(l+1)=index(j)
    index(j)=indext
    jstack=jstack+2
    if (jstack > NSTACK) call nrerror('indexx: NSTACK too small')
    if (r-i+1 >= j-l) then
        istack(jstack)=r
        istack(jstack-1)=i
        r=j-1
    else
        istack(jstack)=j-1
        istack(jstack-1)=l
        l=i
    end if
  end if
end do
CONTAINS
SUBROUTINE icomp_xchg(i,j)
INTEGER(I4B), INTENT(INOUT) :: i,j
INTEGER(I4B) :: swp
if (iarr(j) < iarr(i)) then
  swp=i
  i=j
  j=swp
end if
END SUBROUTINE icomp_xchg
END SUBROUTINE indexx_i4b
```

★ ★ ★

```
SUBROUTINE sort3(arr,slave1,slave2)
USE nrtype; USE nrutil, ONLY : assert_eq
USE nr, ONLY : indexx
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: arr,slave1,slave2
```

Sorts an array arr into ascending order using Quicksort, while making the corresponding rearrangement of the same-size arrays slave1 and slave2. The sorting and rearrangement are performed by means of an index array.

```
INTEGER(I4B) :: ndum
INTEGER(I4B), DIMENSION(size(arr)) :: index
ndum=assert_eq(size(arr),size(slave1),size(slave2),'sort3')
call indexx(arr,index)          Make the index array.
arr=arr(index)                  Sort arr.
slave1=slave1(index)            Rearrange slave1,
slave2=slave2(index)            and slave2.
END SUBROUTINE sort3
```

★ ★ ★

```
FUNCTION rank(index)
USE nrtype; USE nrutil, ONLY : arth
IMPLICIT NONE
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: index
INTEGER(I4B), DIMENSION(size(index)) :: rank
```
Given index as output from the routine indexx, this routine returns a same-size array rank, the corresponding table of ranks.
```
rank(index(:))=arth(1,1,size(index))
END FUNCTION rank
```

⋆ ⋆ ⋆

Just as in the case of sort, where an approximation of the underlying Quicksort partition-exchange algorithm can be captured with the Fortran 90 pack intrinsic, the same can be done with indexx. As before, although it is in principle parallelizable by the compiler, it is likely not competitive with library routines.

```
RECURSIVE SUBROUTINE index_bypack(arr,index,partial)
USE nrtype; USE nrutil, ONLY : array_copy,arth,assert_eq
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: arr
INTEGER(I4B), DIMENSION(:), INTENT(INOUT) :: index
INTEGER, OPTIONAL, INTENT(IN) :: partial
```
Indexes an array arr, i.e., outputs the array index of length N such that arr(index(j)) is in ascending order for $j = 1, 2, \ldots, N$. The method is to apply recursively the Fortran 90 pack intrinsic. This is similar to Quicksort, but allows parallelization by the Fortran 90 compiler. partial is an optional argument that is used only internally on the recursive calls.
```
REAL(SP) :: a
INTEGER(I4B) :: n,k,nl,indext,nerr
INTEGER(I4B), SAVE :: level=0
LOGICAL, DIMENSION(:), ALLOCATABLE, SAVE :: mask
INTEGER(I4B), DIMENSION(:), ALLOCATABLE, SAVE :: temp
if (present(partial)) then
    n=size(index)
else
    n=assert_eq(size(index),size(arr),'indexx_bypack')
    index=arth(1,1,n)
end if
if (n <= 1) RETURN
k=(1+n)/2
call icomp_xchg(index(1),index(k))                  Pivot element is median of first, mid-
call icomp_xchg(index(k),index(n))                      dle, and last.
call icomp_xchg(index(1),index(k))
if (n <= 3) RETURN
level=level+1                                       Keep track of recursion level to avoid
if (level == 1) allocate(mask(n),temp(n))               allocation overhead.
indext=index(k)
a=arr(indext)
mask(1:n) = (arr(index) <= a)                       Which elements move to left?
mask(k) = .false.
call array_copy(pack(index,mask(1:n)),temp,nl,nerr)     Move them.
mask(k) = .true.
temp(nl+2:n)=pack(index,.not. mask(1:n))            Move others to right.
temp(nl+1)=indext
index=temp(1:n)
call index_bypack(arr,index(1:nl),partial=1)        And recurse.
call index_bypack(arr,index(nl+2:n),partial=1)
if (level == 1) deallocate(mask,temp)
level=level-1
```

```
CONTAINS
SUBROUTINE icomp_xchg(i,j)
IMPLICIT NONE
INTEGER(I4B), INTENT(INOUT) :: i,j
  Swap or don't swap integer arguments, depending on the ordering of their corresponding
  elements in an array arr.
INTEGER(I4B) :: swp
if (arr(j) < arr(i)) then
    swp=i
    i=j
    j=swp
end if
END SUBROUTINE icomp_xchg
END SUBROUTINE index_bypack
```

⋆ ⋆ ⋆

```
FUNCTION select(k,arr)
USE nrtype; USE nrutil, ONLY : assert,swap
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: k
REAL(SP), DIMENSION(:), INTENT(INOUT) :: arr
REAL(SP) :: select
    Returns the kth smallest value in the array arr. The input array will be rearranged to have
    this value in location arr(k), with all smaller elements moved to arr(1:k-1) (in arbitrary
    order) and all larger elements in arr(k+1:) (also in arbitrary order).
INTEGER(I4B) :: i,r,j,l,n
REAL(SP) :: a
n=size(arr)
call assert(k >= 1, k <= n, 'select args')
l=1
r=n
do
    if (r-l <= 1) then                  Active partition contains 1 or 2 elements.
        if (r-l == 1) call swap(arr(l),arr(r),arr(l)>arr(r))    Active partition con-
        select=arr(k)                                                tains 2 elements.
        RETURN
    else                                Choose median of left, center, and right elements
        i=(l+r)/2                           as partitioning element a. Also rearrange so
        call swap(arr(i),arr(l+1))          that arr(l) ≤ arr(l+1) ≤ arr(r).
        call swap(arr(l),arr(r),arr(l)>arr(r))
        call swap(arr(l+1),arr(r),arr(l+1)>arr(r))
        call swap(arr(l),arr(l+1),arr(l)>arr(l+1))
        i=l+1                           Initialize pointers for partitioning.
        j=r
        a=arr(l+1)                      Partitioning element.
        do                              Here is the meat.
            do                          Scan up to find element > a.
                i=i+1
                if (arr(i) >= a) exit
            end do
            do                          Scan down to find element < a.
                j=j-1
                if (arr(j) <= a) exit
            end do
            if (j < i) exit             Pointers crossed. Exit with partitioning complete.
            call swap(arr(i),arr(j))        Exchange elements.
        end do
        arr(l+1)=arr(j)                 Insert partitioning element.
        arr(j)=a
        if (j >= k) r=j-1               Keep active the partition that contains the kth
                                            element.
```

```
        if (j <= k) l=i
    end if
end do
END FUNCTION select
```

⋆ ⋆ ⋆

The following routine, `select_inplace`, is renamed from Volume 1's `selip`.

```
FUNCTION select_inplace(k,arr)
USE nrtype
USE nr, ONLY : select
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: k
REAL(SP), DIMENSION(:), INTENT(IN) :: arr
REAL(SP) :: select_inplace
    Returns the kth smallest value in the array arr, without altering the input array. In Fortran
    90's assumed memory-rich environment, we just call select in scratch space.
REAL(SP), DIMENSION(size(arr)) :: tarr
tarr=arr
select_inplace=select(k,tarr)
END FUNCTION select_inplace
```

f90 Volume 1's `selip` routine uses an entirely different algorithm, for the purpose of avoiding any additional memory allocation beyond that of the input array. Fortran 90 presumes a richer memory environment, so `select_inplace` simply does the obvious (destructive) selection in scratch space. You can of course use the old `selip` if your in-core or in-cache memory is at a premium.

```
FUNCTION select_bypack(k,arr)
USE nrtype; USE nrutil, ONLY : array_copy,assert,swap
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: k
REAL(SP), DIMENSION(:), INTENT(INOUT) :: arr
REAL(SP) :: select_bypack
    Returns the kth smallest value in the array arr. The input array will be rearranged to have
    this value in location arr(k), with all smaller elements moved to arr(1:k-1) (in arbitrary
    order) and all larger elements in arr(k+1:) (also in arbitrary order). This implementation
    allows parallelization in the Fortran 90 pack intrinsic.
LOGICAL, DIMENSION(size(arr)) :: mask
REAL(SP), DIMENSION(size(arr)) :: temp
INTEGER(I4B) :: i,r,j,l,n,nl,nerr
REAL(SP) :: a
n=size(arr)
call assert(k >= 1, k <= n, 'select_bypack args')
l=1                                                    Initial left and right bounds.
r=n
do                                                     Keep partitioning until desired el-
    if (r-l <= 1) exit                                     ement is found.
    i=(l+r)/2
    call swap(arr(l),arr(i),arr(l)>arr(i))             Pivot element is median of first,
    call swap(arr(i),arr(r),arr(i)>arr(r))                 middle, and last.
    call swap(arr(l),arr(i),arr(l)>arr(i))
    a=arr(i)
    mask(l:r) = (arr(l:r) <= a)                        Which elements move to left?
    mask(i) = .false.
    call array_copy(pack(arr(l:r),mask(l:r)),temp(l:),nl,nerr)      Move them.
    j=l+nl
```

```
    mask(i) = .true.
    temp(j+1:r)=pack(arr(l:r),.not. mask(l:r))        Move others to right.
    temp(j)=a
    arr(l:r)=temp(l:r)
    if (k > j) then                                   Reset bounds to whichever side
        l=j+1                                             has the desired element.
    else if (k < j) then
        r=j-1
    else
        l=j
        r=j
    end if
end do
if (r-l == 1) call swap(arr(l),arr(r),arr(l)>arr(r))      Case of only two left.
select_bypack=arr(k)
END FUNCTION select_bypack
```

The above routine `select_bypack` is parallelizable, but as discussed above (`sort_bypack`, `index_bypack`) it is generally not very efficient.

⋆ ⋆ ⋆

The following routine, `select_heap`, is renamed from Volume 1's `hpsel`.

```
SUBROUTINE select_heap(arr,heap)
USE nrtype; USE nrutil, ONLY : nrerror,swap
USE nr, ONLY : sort
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: arr
REAL(SP), DIMENSION(:), INTENT(OUT) :: heap
```

Returns in `heap`, an array of length M, the largest M elements of the array `arr` of length N, with `heap(1)` guaranteed to be the the Mth largest element. The array `arr` is not altered. For efficiency, this routine should be used only when $M \ll N$.

```
INTEGER(I4B) :: i,j,k,m,n
m=size(heap)
n=size(arr)
if (m > n/2 .or. m < 1) call nrerror('probable misuse of select_heap')
heap=arr(1:m)
call sort(heap)                       Create initial heap by overkill! We assume m ≪ n.
do i=m+1,n                            For each remaining element...
    if (arr(i) > heap(1)) then            Put it on the heap?
        heap(1)=arr(i)
        j=1
        do                            Sift down.
            k=2*j
            if (k > m) exit
            if (k /= m) then
                if (heap(k) > heap(k+1)) k=k+1
            end if
            if (heap(j) <= heap(k)) exit
            call swap(heap(k),heap(j))
            j=k
        end do
    end if
end do
END SUBROUTINE select_heap
```

⋆ ⋆ ⋆

```
FUNCTION eclass(lista,listb,n)
USE nrtype; USE nrutil, ONLY : arth,assert_eq
IMPLICIT NONE
INTEGER(I4B), DIMENSION(:), INTENT(IN) :: lista,listb
INTEGER(I4B), INTENT(IN) :: n
INTEGER(I4B), DIMENSION(n) :: eclass
```

Given M equivalences between pairs of `n` individual elements in the form of the input arrays `lista` and `listb` of length M, this routine returns in an array of length `n` the number of the equivalence class of each of the `n` elements, integers between 1 and `n` (not all such integers used).

```
INTEGER :: j,k,l,m
m=assert_eq(size(lista),size(listb),'eclass')
eclass(1:n)=arth(1,1,n)             Initialize each element its own class.
do l=1,m                            For each piece of input information...
    j=lista(l)
    do                              Track first element up to its ancestor.
        if (eclass(j) == j) exit
        j=eclass(j)
    end do
    k=listb(l)
    do                              Track second element up to its ancestor.
        if (eclass(k) == k) exit
        k=eclass(k)
    end do
    if (j /= k) eclass(j)=k         If they are not already related, make them so.
end do
do j=1,n                            Final sweep up to highest ancestors.
    do
        if (eclass(j) == eclass(eclass(j))) exit
        eclass(j)=eclass(eclass(j))
    end do
end do
END FUNCTION eclass

FUNCTION eclazz(equiv,n)
USE nrtype; USE nrutil, ONLY : arth
IMPLICIT NONE
INTERFACE
    FUNCTION equiv(i,j)
    USE nrtype
    IMPLICIT NONE
    LOGICAL(LGT) :: equiv
    INTEGER(I4B), INTENT(IN) :: i,j
    END FUNCTION equiv
END INTERFACE
INTEGER(I4B), INTENT(IN) :: n
INTEGER(I4B), DIMENSION(n) :: eclazz
```

Given a user-supplied logical function `equiv` that tells whether a pair of elements, each in the range `1...n`, are related, return in an array of length `n` equivalence class numbers for each element.

```
INTEGER :: i,j
eclazz(1:n)=arth(1,1,n)
do i=2,n                                    Loop over first element of all pairs.
    do j=1,i-1                              Loop over second element of all pairs.
        eclazz(j)=eclazz(eclazz(j))         Sweep it up this much.
        if (equiv(i,j)) eclazz(eclazz(eclazz(j)))=i
          Good exercise for the reader to figure out why this much ancestry is necessary!
    end do
end do
do i=1,n                                    Only this much sweeping is needed finally.
    eclazz(i)=eclazz(eclazz(i))
end do
END FUNCTION eclazz
```

CITED REFERENCES AND FURTHER READING:

Knuth, D.E. 1973, *Sorting and Searching*, vol. 3 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), §5.2.5. [1]