```
SUBROUTINE rank(n,indx,irank)
INTEGER n,indx(n),irank(n)
    Given indx(1:n) as output from the routine indexx, this routine returns an array irank(1:n),
    the corresponding table of ranks.
INTEGER j
do 11 j=1,n
    irank(indx(j))=j
enddo 11
return
END
```

Figure 8.4.1 summarizes the concepts discussed in this section.

8.5 Selecting the Mth Largest

Selection is sorting's austere sister. (Say *that* five times quickly!) Where sorting demands the rearrangement of an entire data array, selection politely asks for a single returned value: What is the kth smallest (or, equivalently, the $m = N+1-k$th largest) element out of N elements? The fastest methods for selection do, unfortunately, rearrange the array for their own computational purposes, typically putting all smaller elements to the left of the kth, all larger elements to the right, and scrambling the order within each subset. This side effect is at best innocuous, at worst downright inconvenient. When the array is very long, so that making a scratch copy of it is taxing on memory, or when the computational burden of the selection is a negligible part of a larger calculation, one turns to selection algorithms without side effects, which leave the original array undisturbed. Such *in place* selection is slower than the faster selection methods by a factor of about 10. We give routines of both types, below.

The most common use of selection is in the statistical characterization of a set of data. One often wants to know the median element in an array, or the top and bottom quartile elements. When N is odd, the median is the kth element, with $k = (N+1)/2$. When N is even, statistics books define the median as the arithmetic mean of the elements $k = N/2$ and $k = N/2 + 1$ (that is, $N/2$ from the bottom and $N/2$ from the top). If you accept such pedantry, you must perform two separate selections to find these elements. For $N > 100$ we usually define $k = N/2$ to be the median element, pedants be damned.

The fastest general method for selection, allowing rearrangement, is *partitioning*, exactly as was done in the Quicksort algorithm (§8.2). Selecting a "random" partition element, one marches through the array, forcing smaller elements to the left, larger elements to the right. As in Quicksort, it is important to optimize the inner loop, using "sentinels" (§8.2) to minimize the number of comparisons. For sorting, one would then proceed to further partition both subsets. For selection, we can ignore one subset and attend only to the one that contains our desired kth element. Selection by partitioning thus does not need a stack of pending operations, and its operations count scales as N rather than as $N \log N$ (see [1]). Comparison with `sort` in §8.2 should make the following routine obvious:

```
      FUNCTION select(k,n,arr)
      INTEGER k,n
      REAL select,arr(n)
        Returns the kth smallest value in the array arr(1:n). 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..n] (also in arbitrary order).
      INTEGER i,ir,j,l,mid
      REAL a,temp
      l=1
      ir=n
1     if(ir-l.le.1)then                 Active partition contains 1 or 2 elements.
        if(ir-l.eq.1)then               Active partition contains 2 elements.
          if(arr(ir).lt.arr(l))then
            temp=arr(l)
            arr(l)=arr(ir)
            arr(ir)=temp
          endif
        endif
        select=arr(k)
        return
      else
        mid=(l+ir)/2                    Choose median of left, center, and right elements as par-
        temp=arr(mid)                     titioning element a. Also rearrange so that arr(l) ≤
        arr(mid)=arr(l+1)                 arr(l+1), arr(ir) ≥ arr(l+1).
        arr(l+1)=temp
        if(arr(l).gt.arr(ir))then
          temp=arr(l)
          arr(l)=arr(ir)
          arr(ir)=temp
        endif
        if(arr(l+1).gt.arr(ir))then
          temp=arr(l+1)
          arr(l+1)=arr(ir)
          arr(ir)=temp
        endif
        if(arr(l).gt.arr(l+1))then
          temp=arr(l)
          arr(l)=arr(l+1)
          arr(l+1)=temp
        endif
        i=l+1                           Initialize pointers for partitioning.
        j=ir
        a=arr(l+1)                      Partitioning element.
3       continue                        Beginning of innermost loop.
          i=i+1                         Scan up to find element > a.
        if(arr(i).lt.a)goto 3
4       continue
          j=j-1                         Scan down to find element < a.
        if(arr(j).gt.a)goto 4
        if(j.lt.i)goto 5                Pointers crossed. Exit with partitioning complete.
        temp=arr(i)                     Exchange elements.
        arr(i)=arr(j)
        arr(j)=temp
        goto 3                          End of innermost loop.
5       arr(l+1)=arr(j)                 Insert partitioning element.
        arr(j)=a
        if(j.ge.k)ir=j-1                Keep active the partition that contains the kth element.
        if(j.le.k)l=i
      endif
      goto 1
      END
```

In-place, nondestructive, selection is conceptually simple, but it requires a lot of bookkeeping, and it is correspondingly slower. The general idea is to pick some number M of elements at random, to sort them, and then to make a pass through the array *counting* how many elements fall in each of the $M + 1$ intervals defined by these elements. The kth largest will fall in one such interval — call it the "live" interval. One then does a second round, first picking M random elements in the live interval, and then determining which of the new, finer, $M + 1$ intervals all presently live elements fall into. And so on, until the kth element is finally localized within a single array of size M, at which point direct selection is possible.

How shall we pick M? The number of rounds, $\log_M N = \log_2 N / \log_2 M$, will be smaller if M is larger; but the work to locate each element among $M + 1$ subintervals will be larger, scaling as $\log_2 M$ for bisection, say. Each round requires looking at all N elements, if only to find those that are still alive, while the bisections are dominated by the N that occur in the first round. Minimizing $O(N \log_M N) + O(N \log_2 M)$ thus yields the result

$$M \sim 2^{\sqrt{\log_2 N}} \tag{8.5.1}$$

The square root of the logarithm is so slowly varying that secondary considerations of machine timing become important. We use $M = 64$ as a convenient constant value.

Two minor additional tricks in the following routine, `selip`, are (i) augmenting the set of M random values by an $M + 1$st, the arithmetic mean, and (ii) choosing the M random values "on the fly" in a pass through the data, by a method that makes later values no less likely to be chosen than earlier ones. (The underlying idea is to give element $m > M$ an M/m chance of being brought into the set. You can prove by induction that this yields the desired result.)

```
      FUNCTION selip(k,n,arr)
      INTEGER k,n,M
      REAL selip,arr(n),BIG
      PARAMETER (M=64,BIG=1.E30)
        Returns the kth smallest value in the array arr(1:n). The input array is not altered.
C     USES shell
      INTEGER i,j,jl,jm,ju,kk,mm,nlo,nxtmm,isel(M+2)
      REAL ahi,alo,sum,sel(M+2)
      if(k.lt.1.or.k.gt.n.or.n.le.0) pause 'bad input to selip'
      kk=k
      ahi=BIG
      alo=-BIG
1     continue                              Main iteration loop, until desired element is isolated.
        mm=0
        nlo=0
        sum=0.
        nxtmm=M+1
        do 11 i=1,n                         Make a pass through the whole array.
          if(arr(i).ge.alo.and.arr(i).le.ahi)then     Consider only elements in the cur-
            mm=mm+1                                       rent brackets.
            if(arr(i).eq.alo) nlo=nlo+1               In case of ties for low bracket.
            if(mm.le.M)then                 Statistical procedure for selecting m in-range elements
              sel(mm)=arr(i)                    with equal probability, even without knowing in
            else if(mm.eq.nxtmm)then            advance how many there are!
              nxtmm=mm+mm/M
              sel(1+mod(i+mm+kk,M))=arr(i)            The mod function provides a some-
            endif                                         what random number.
            sum=sum+arr(i)
```

```
        endif
    enddo 11
    if(kk.le.nlo)then                Desired element is tied for lower bound; return it.
        selip=alo
        return
    else if(mm.le.M)then             All in-range elements were kept. So return answer by
        call shell(mm,sel)               direct method.
        selip=sel(kk)
        return
    endif                            Augment selected set by mean value (fixes degenera-
    sel(M+1)=sum/mm                      cies), and sort it.
    call shell(M+1,sel)
    sel(M+2)=ahi
    do 12 j=1,M+2                    Zero the count array.
        isel(j)=0
    enddo 12
    do 13 i=1,n                      Make another pass through the whole array.
        if(arr(i).ge.alo.and.arr(i).le.ahi)then      For each in-range element..
            jl=0
            ju=M+2
2           if(ju-jl.gt.1)then       ...find its position among the select by bisection...
                jm=(ju+jl)/2
                if(arr(i).ge.sel(jm))then
                    jl=jm
                else
                    ju=jm
                endif
            goto 2
            endif
            isel(ju)=isel(ju)+1      ...and increment the counter.
        endif
    enddo 13
    j=1                              Now we can narrow the bounds to just one bin, that
3   if(kk.gt.isel(j))then                is, by a factor of order m.
        alo=sel(j)
        kk=kk-isel(j)
        j=j+1
    goto 3
    endif
    ahi=sel(j)
goto 1
END
```

Approximate timings: `selip` is about 10 times slower than `select`. Indeed, for N in the range of $\sim 10^5$, `selip` is about 1.5 times slower than a full sort with `sort`, while `select` is about 6 times faster than `sort`. You should weigh time against memory and convenience carefully.

Of course neither of the above routines should be used for the trivial cases of finding the largest, or smallest, element in an array. Those cases, you code by hand as simple `do` loops. There are also good ways to code the case where k is modest in comparison to N, so that extra memory of order k is not burdensome. An example is to use the method of Heapsort (§8.3) to make a single pass through an array of length N while saving the m *largest* elements. The advantage of the heap structure is that only $\log m$, rather than m, comparisons are required every time a new element is added to the candidate list. This becomes a real savings when $m > O(\sqrt{N})$, but it never hurts otherwise and is easy to code. The following program gives the idea.

```
SUBROUTINE hpsel(m,n,arr,heap)
INTEGER m,n
```

```
      REAL arr(n),heap(m)
C     USES sort
        Returns in heap(1:m) the largest m elements of the array arr(1: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 ≪ n.
      INTEGER i,j,k
      REAL swap
      if (m.gt.n/2.or.m.lt.1) pause 'probable misuse of hpsel'
      do 11 i=1,m
          heap(i)=arr(i)
      enddo 11
      call sort(m,heap)                 Create initial heap by overkill! We assume m ≪ n.
      do 12 i=m+1,n                     For each remaining element...
          if(arr(i).gt.heap(1))then     Put it on the heap?
              heap(1)=arr(i)
              j=1
1             continue                  Sift down.
                  k=2*j
                  if(k.gt.m)goto 2
                  if(k.ne.m)then
                      if(heap(k).gt.heap(k+1))k=k+1
                  endif
                  if(heap(j).le.heap(k))goto 2
                  swap=heap(k)
                  heap(k)=heap(j)
                  heap(j)=swap
                  j=k
              goto 1
2             continue
          endif
      enddo 12
      return
      end
```

CITED REFERENCES AND FURTHER READING:

Sedgewick, R. 1988, *Algorithms*, 2nd ed. (Reading, MA: Addison-Wesley), pp. 126ff. [1]

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

8.6 Determination of Equivalence Classes

A number of techniques for sorting and searching relate to data structures whose details are beyond the scope of this book, for example, trees, linked lists, etc. These structures and their manipulations are the bread and butter of computer science, as distinct from numerical analysis, and there is no shortage of books on the subject.

In working with experimental data, we have found that one particular such manipulation, namely the determination of equivalence classes, arises sufficiently often to justify inclusion here.

The problem is this: There are N "elements" (or "data points" or whatever), numbered $1, \ldots, N$. You are given pairwise information about whether elements are in the same *equivalence class* of "sameness," by whatever criterion happens to be of interest. For example, you may have a list of facts like: "Element 3 and element 7 are in the same class; element 19 and element 4 are in the same class; element 7 and element 12 are in the same class, ..." Alternatively, you may have a procedure, given the numbers of two elements